
Copyright 2007 Association for Computing Machinery (ACM)
ABSTRACT
Declarative approaches to specifying model-model transformation are an attractive approach because they can offer implicit source model traversal, automatic traceability management, implicit target object creation, and implicit rule ordering. However, when proposing such a declarative logic-based transformation language, there are two common objections. One is programmer unfamiliarity with declarative style, and the other is that of perceived performance problems. In this paper we address these issues, discussing the design of specific features of the Tefkat transformation engine intended to facilitate writing and debugging declarative transformation specifications, and describing important implementation techniques used to avoid performance problems.

Categories and Subject Descriptors

General Terms
Performance, Design, Languages.

Keywords
Model transformation, Model Driven Development, MDA, MOF

1. Introduction
The Model Driven Architecture (MDA) promises to deliver rapid and reliable application development. The key component to realising this vision is the ability to specify and perform model-to-model transformations. The Object Management Group (OMG) has recently standardised MOF Query/View/Transformation (QVT) [1] to support model-to-model transformations. The QVT standard offers a declarative approach, an imperative approach, and a hybrid approach to transformation specification. This trifurcation has been criticised as being "three sub-standards" which does not fulfil the goal of interoperability [2].

This unease reflects the fundamentally different approach of declarative and imperative programming. A declarative program describes the goal explicitly and leaves the algorithm implicit, whereas imperative programming describes computation in terms of a program state and statements that change the program state. Winograd [3] distinguishes these as:

"(1) Program specification. A formal structure which can be interpreted as a set of instructions for a given machine. This is the imperative style of traditional programming languages.

(2) Result specification. A process-independent specification of the relationships between the inputs (or initial state), internal variables, and outputs (or resulting state) of the program."

In the context of model transformation, a declarative approach focuses on the relationships between the source and target models whereas the imperative approach focuses on the steps to construct the target models from the source models.

Although the benefits of a declarative approach were acknowledged by many involved in the specification of the QVT standard, concerns about the practicality of such an approach lead to the inclusion of imperative and hybrid alternative languages.

The goal of this paper is to demonstrate that a purely declarative approach to model transformation is practical and achievable; delivering the advantages without the disadvantages.

1.1 Advantages of the declarative approach
As observed by Mens et al [4], declarative approaches to specifying model-model transformation are an attractive approach because they can offer implicit source model traversal, automatic traceability management, implicit target object creation, and implicit rule ordering. In addition, declarative languages lend themselves to automatic analysis and hence provide opportunities for optimisation and specialised tool support.

We believe that a declarative approach to model-to-model transformation provides the following benefits:
Implicit source model traversal: pattern matching is done by transformation engine rather than explicitly coded using the model API.

Automatic traceability management: relationship between a target object and the source objects that lead to its creation is automatically recorded during the transformation execution.

Implicit target object creation: instead of a new Object statement, the existence of a target object is asserted by one or more constraints. This simplifies the case where multiple conditions independently require the same target object.

Implicit rule ordering: the programmer is not forced to impose a total order on rule execution.

Side-effect free (idempotent): rules can be more easily understood in isolation and in combination.

Automatic analysis: simpler and greater opportunities for optimisation & tool support.

Programmer productivity: programs are shorter and more direct in their expression.

1.2 Disadvantages of the declarative approach
Common criticisms of the declarative approach include:

- Unfamiliar style: often exacerbated by unfamiliar and terse concrete syntax.
- Performance problems: often interpreted, mismatch with hardware requires optimisation to bridge the gap efficiently. Less control of algorithms used.
- Lack of libraries and tool support.
- Sometimes it is easier to explain an algorithm than to describe the goal.
- Debugging is often more difficult than with imperative languages.

1.3 Structure of paper
The paper is organised as follows. We begin by enumerating the design principles of a declarative QVT engine developed at DSTC known as Tefkat [5]. We then discuss specific features of the Tefkat language that are intended to facilitate writing and debugging declarative transformation specifications by programmers unfamiliar with the declarative paradigm. Next we describe aspects of the Tefkat implementation that enable the engine to efficiently perform model transformations. We then discuss several aspects related to tool support for writing transformations. Finally we conclude with a discussion of further enhancements of Tefkat to provide even better support for declarative model transformation.

Throughout the paper we use fragments of the ever-popular Object-Relational Mapping example [6,7,8,9,10,11] to illustrate key points.

2. Tefkat Design
Tefkat was designed and implemented at DSTC between 2003 and 2005. It was preceded by a series of experimental model transformation engines going back to 2000 [12]. Many of the design decisions of Tefkat were therefore informed by this prior research and experience, in particular the choice of a declarative logic-based approach over an imperative approach.

The following key design choices were made (clearly these are not orthogonal decisions):

- it would be a declarative logic-based language, but would not require full unification; specifically it would not support function symbols or the construction of Prolog-like terms.
- it would be implemented in Java.
- rules should “fire” when patterns are matched rather than as a result of explicit rule invocation; the engine would determine appropriate rule execution order.
- target objects do not have to be created by a single rule; different rules can determine different aspects of the same target object based on separation of concerns.
- patterns of target objects (or parts thereof) can be created in a single rule.
- multiplicities, optional attributes, and ordering should all be easily managed (not forcing explicit iteration, conditions, or additional rules).
- variables should be dynamically typed.
- generic (meta-model/type independent) rules should be possible and easily expressible.
- we were designing an abstract syntax (meta-model) which has a different set of design criteria to the design of a concrete syntax.

We chose a declarative logic-based approach because our earlier experiments with an imperative approach that highlighted the complexities intrinsic to explicit object creation and the difficulty of separating the constraints that related source and target models from the explicit ordering of rules and model traversal. Implicit source model traversal reduces coupling between rules and the source meta-model which enhances rule readability, reusability, and reduces the impact of transformation and meta-model evolution [13].

Tefkat is not intended to be a general purpose programming language, but is designed specifically for specifying model-to-model transformations. In this context, unification is used as part of pattern matching over a finite set of source objects. Avoiding full unification significantly improves the performance of the underlying engine. Adopting a declarative approach also allowed us to ensure a consistent and clean semantics for Tefkat, derived from the (two-valued) well-founded semantics [14] of an equivalent logic program.

The choice of Java as an implementation language for a declarative language like Tefkat might not seem obvious, especially since some of our earlier declarative transformation tools had been built on a variety of Prolog engines. We found that emulating MOF semantics in a Prolog engine and the overhead of full unification were key aspects limiting the performance of our prototypes [12]. This was compounded by the mismatch between model transformation having an all-answers semantics rather than Prolog’s first-answer semantics. We chose Java because of the availability of a Java-based MOF 2.0 [15], its ability to be deployed over multiple platforms, and its widespread use in the emerging community of MDA users. The availability of the Java-based MOF 2.0 implementation, EMF [16], and its integration into the Eclipse IDE opened significant opportunities for Tefkat to be a complete tool including XMI import/export, a syntax-aware
editor, a source-level debugger, and the ability to provide our own Eclipse extension points for others to further extend Tefkat.

3. Tefkat Overview
A Tefkat transformation consists of an unordered set of rules. Figure 1 illustrates an example of a Tefkat rule.

```
RULE ClassAndTable
  FORALL Class C
  WHERE C.is_persistent = true
  AND C.name = N
  MAKE Table T FROM class2table(C)
  SET T.name = N;
```

Figure 1. Example of a Tefkat rule

A Tefkat rule consists of a source pattern and a target pattern (see Figure 1). Here, the source pattern matches all instances of the type Class for which is_persistent is true, binding the variable C to the object reference representing the Class and the variable N to the name of the class (a string). Note that variables are dynamically typed. The target pattern asserts the existence of a Table T corresponding to each Class C and having the same name. If there are no persistent Classes, the source pattern matches nothing and no Tables are created by this rule.

To find and match the Classes it is not necessary to navigate from some root element as is common in languages with explicit rule invocation such as XSLT [17].

The target pattern is an assertion, not an explicit object creation statement. This allows multiple rules to assert the existence of the same object with each rule providing different detail, thus supporting separation of concerns. It is even possible for such rules to specify different types (related by inheritance) for the same object. It is the combined set of assertions that are used to construct the final set of target objects (the “make it so” [18] step).

The FROM clause creates a named relationship (class2table in Figure 1) between source and target objects corresponding to the automatic traceability information. This enables multiple rules to reference the same target object. In addition, tracking classes allow programmer-specified traceability relationships.

Patterns are not restricted to matching source objects and their attributes but can also invoke operations defined on source objects. Naturally only side-effect free operations should be invoked.

As transformation rules often include common expressions it is desirable to be able to factor these out into a set of sharable PATTERNs and TEMPLATEs as illustrated in Figure 2. PATTERNs are used to capture common source matching constraints while TEMPLATEs are re-usable target assertions.

Figure 2 shows an equivalent transformation to that in Figure 1 but using a PATTERN and TEMPLATE. Note that both PATTERNs and TEMPLATEs may be defined recursively, for example to navigate tree or graph structures such as an inheritance hierarchy.

```
PATTERN persistentClass(C, N)
  WHERE Class C
  AND C.is_persistent = true
  AND C.name = N;

TEMPLATE tableFromClass(C, N)
  MAKE Table T FROM class2table(C)
  SET T.name = N;

RULE PersistentClassAndTable
  WHERE persistentClass(C, N)
  MAKE tableFromClass(C, N);
```

Figure 2. Example of a Tefkat PATTERN and TEMPLATE

Another form of re-use involves abstract rules and rule extension, analogous to abstract classes and class extension. This is illustrated in Figure 3.

```
ABSTRACT RULE ClassAndTable(C)
  FORALL Class C
  WHERE C.is_persistent = true
  AND C.name = N
  MAKE Table T FROM class2table(C)
  SET T.name = N;

RULE PersistentClassAndTable(C)
  WHERE C.is_persistent = true;
```

Figure 3. Example use of a Tefkat abstract rule.

In Figure 3 the abstract rule ClassAndTable can encapsulate the construction of a Table from a Class, setting the Table’s properties based on the Class’s properties. However, this rule does not fire but is incorporated into any extending rules such as PersistentClassAndTable that define the circumstances in which a table should be constructed from a Class, in this case, when the Class is persistent.

Abstract rules and rule extension provide another structuring mechanism for achieving separation of concerns. In particular, rules that process subtypes are often best written as extensions of a rule that processes their common supertype.

4. Tefkat Concrete Syntax
An important design decision for Tefkat was the decoupling of the transformation model (abstract syntax) from the concrete syntax. This allows a transformation to be expressed in a number of different ways and decouples the implementation of the engine from a parser of any specific concrete syntax, even allowing tools to directly generate the abstract syntax without using a concrete syntax or parser.

Even within the concrete syntax supported by the default Tefkat parser, there is not a one-to-one mapping between the model and the concrete syntax. This allows us to offer syntactic alternatives and convenient syntactic defaults thus eliminating boilerplate syntax and textual noise.

Our default concrete syntax derives from that of SQL in order to be familiar to traditional imperative programmers who might otherwise baulk at a more Prolog-like syntax. A drawback of this choice is a somewhat verbose and irregular syntax. As a consequence we have introduced alternative syntactic forms that are more compact.

Figure 4 shows an example compound expression followed by an equivalent expression using object literal syntax; nested terms are
evaluated in the context of their containing term, that is Name is the name of the ClassCls. Nesting can be arbitrarily deep and is not restricted to containment relationships, allowing large object graphs to be matched. Many users find this syntax more compact, better structured, and vastly more readable.

Class Cls AND
Cls.attrs = Attr AND
Attribute Attr AND
Attr.type = PT AND
PrimitiveDataType PT AND
Attr.name = Name AND
Attr.is_primary = IsKey

Class Cls {
    attrs: Attribute Attr {
        type: PrimitiveDataType _PT;
        name: Name;
        is_primary: IsKey;
    };
}

Figure 4. Equivalent expressions using traditional syntax, and then object literal syntax

The alert reader will also have noticed the omission of the FROM clause in the MAKE term in Figure 3. When omitted, a default clause is implied that is specific to the rule and is based on the set of outermost variables in the FORALL (C in this example) as this corresponds with the usual programmer thought process (“for each this and this, make that”).

Other examples of alternate syntax are FORALL versus WHERE and MAKE versus SET. The underlying abstract syntax (model) makes no distinction between FORALL and WHERE or between MAKE and SET. The distinction exists purely in the concrete syntax and reflects/guides the programmer’s thought processes. The FORALL conveys the sense of iterating over a set of source objects whereas WHERE suggests a selection or filtering of those objects. Similarly, MAKE is used to assert the existence of objects whereas SET is used to assert properties of these objects.

Objects can have multi-valued properties, so there must be a way to access the values of these properties both individually and collectively as illustrated in Figure 5.

RULE processAnAttribute
    FORALL Class C
    WHERE C.attr = A
    MAKE Something S FROM s(C, A);
RULE processAllAttributes
    FORALL Class C
    WHERE C.attr{} = A
    MAKE Something S FROM s(C, A);

Figure 5. Example of working with multi-valued properties

Rule processAnAttribute in Figure 5 illustrates the binding of the variable A to each of the many attributes of Class C. The concrete syntax makes no distinction between multi-valued and single valued attributes. Thus, for a single Class with n attributes, n Somethings would be created. In contrast, the rule processAllAttributes illustrates the binding of the variable A to all of the attributes of Class C (indicated by the use of {} decoration). Thus, A is bound to an ordered collection of Attributes and for a single Class with any number of attributes, only one Something is created. The simpler undecorated syntax is used for the far more common case in which the programmer wishes to match individual values rather than collections of values.

Multi-valued properties can be defined to be ordered or unordered. Therefore Tefkat must provide a means to both match and specify an explicit order for an ordered multi-valued property, as illustrated in Figure 6. On the target side, this is done by asserting a partial order for the target property’s values, possibly over several rules. Tefkat then chooses an arbitrary total order that satisfies this partial order.

RULE orderColumns
    FORALL Class C
    WHERE Attr1 BEFORE Attr2 in C.attr
    ...
    SET Col1 BEFORE Col2 in T.columns;

Figure 6. Example Tefkat rule both matching and asserting the order of multi-valued properties.

Here in Figure 6 we see the columns of the table being placed in the same order as their corresponding attributes are in the corresponding class. Typically ordering in a target is derived from an order that already exists in the source. Because ordering needs to talk about two target objects, it is usual for ordering to be specified in a separate rule from the rule(s) that created these objects. This is an example of separation of concerns into separate Tefkat rules which makes for a more modular and hence more readable transformation.

Tefkat has a number of advanced features including reflection and indirection both of which are illustrated in Figure 7 below.

RULE copyObjects
    FORALL _ Src
    MAKE $Src.eClass() Tgt

Figure 7. Example of a Meta-level rule

The rule copyObjects in Figure 7 illustrates the syntactic use of underscore to represent a virtual Any class; the variable Src will be bound to each source object regardless of type. While MOF includes the class MOF::Object that all MOF objects are instances of, this occurs at the implementation level, and MOF::Object is not actually a superclass of all classes. While we could have blurred this fact in Tefkat, it would have made writing transformations of MOF meta-models very difficult. Using a virtual common superclass in place of MOF::Object avoids these problems. The same issues apply to EMF models with respect to ECore and EObject.

The MAKE clause illustrates both indirection and reflection. The use of the $ syntax introduces an expression where a literal would normally appear (indirection). The invocation of the eClass() method illustrates reflection in which an object’s metadata (its class) is queried.

The overall effect of this rule is to create a target object corresponding to each source object and of the same class. Note
that an additional rule, also using reflection, would be required to copy the values of source object properties.

Further details on Tefkat’s concrete syntax can be found in [8].

5. Engine Implementation

In implementing the Tefkat transformation engine we took the approach of first choosing the simplest and most direct mechanisms (a logic programming interpreter using a form of SLDNF resolution), then, as performance issues have arisen, we used advanced techniques to avoid these problems. It was due to specific properties resulting from the design of the language that we were able to use some of these techniques.

5.1 Order of rule evaluation

The most important property of Tefkat is that the meaning of a transformation specification is not dependent on a specific evaluation order derived from the order the rules are specified. Instead, it is dependent on semantic dependencies between rules, based on the dependency analysis of the LINKS and LINKING clauses. Consider, for example,

\[
\begin{align*}
\text{RULE R1} & \quad \text{LINKING T} \ldots; \\
\text{RULE R2} & \quad \text{WHERE T LINKS} \ldots;
\end{align*}
\]

The evaluation of a rule that queries (LINKS) a tracking class cannot be considered complete until all the rules that could create (LINKING) an instance of that tracking class have themselves completed. In our example, R2 cannot be considered completely evaluated while there remains the possibility that further evaluation of R1 may yield a new instance of T. As there might be cyclic dependencies between R1 and R2, it may be necessary to interleave the evaluation of rules R1 and R2.

The situation is more complex where there is a negated query (NOT-LINKS) on a tracking class. For example,

\[
\begin{align*}
\text{RULE R3} & \quad \text{WHERE NOT (T LINKS \ldots)}
\end{align*}
\]

Here the NOT-LINKS clause of R3 cannot be evaluated until all rules that might create an instance of tracking class T (i.e. R1) have been completely evaluated. Thus the evaluation of R1 and R3 cannot be interleaved. Therefore, by static analysis of the LINKING, LINKS and NOT-LINKS within the rules, the engine can devise an execution order (known as stratification or level mapping [14]) that is both correct and complete. Each stratum/level is evaluated completely, in strict sequence. Within each stratum, rule evaluation may be interleaved.

If there is a cyclic dependency involving NOT-LINKS, the rules cannot be stratified. This means that there is more than one way to evaluate the rules, with each evaluation strategy leading to a different set of targets, and there is no natural/intuitive way to choose one evaluation over another, thus leading to a non-deterministic outcome. Our experience indicates that this kind of non-determinism is undesirable in a model transformation and thus Tefkat rejects transformations containing rules that cannot be stratified.

5.2 Incremental evaluation

The interleaving of evaluation of rules within a stratum can be achieved in two ways. The simplest approach is to execute each of the rules in turn to completion. Then if any of them creates tracking class instances, all of the rules are re-executed using the updated set of tracking class instances. This process of re-execution repeats until no new tracking class instances are created (a fixpoint has been reached).

This re-execution strategy was easy to implement. For many transformations, only one or two iterations are required and the performance of the engine was quite acceptable. However, more complex transformations perform badly with this simple approach.

Re-execution of the rules involves rebuilding the entire evaluation tree, repeating work done in previous executions. This is clearly inefficient. The incremental evaluation strategy reuses the evaluation tree from the previous iteration. By remembering which nodes in the tree involved querying a tracking class, a new branch can be added to the tree, corresponding to each new tracking class instance. Thus the evaluation tree is grown incrementally, avoiding unnecessary re-computation.

5.3 Caching of patterns and templates

Each evaluation of a pattern (or template) constructs an evaluation tree for that pattern invocation. Performance analysis revealed that some patterns were invoked with the same parameters many times, needlessly reconstructing identical evaluation trees. By caching these pattern evaluation trees (keyed on the input parameters), this unnecessary computation was avoided, markedly improving performance.

More sophisticated caching strategies exist, but the implementation costs do not appear to be justified by the performance improvement.

5.4 Order of term evaluation

Within a rule, the order in which terms are evaluated is determined by the engine based on several simple criteria intended to minimise branching of the evaluation tree since the number of nodes in the tree correlates with both the space and time cost of transformation evaluation. Thus terms with at most a single solution are selected before those with potentially multiple solutions, and terms that query the source are selected before those that affect the target.

However, it may happen that a term is selected for evaluation before sufficient information is available (not enough variables have been bound to a value) in which case evaluation of this term is then delayed in anticipation that evaluation of other terms will provide the required information (variable bindings). For example, consider attempting to evaluate the following from left to right: \( Y = X + 1 \) AND \( X = O.foo \). If \( X \) is unbound, then the first term, \( Y = X + 1 \) cannot be evaluated so it is delayed. Evaluation of \( X = O.foo \) (assuming \( O \) is bound) then binds \( X \) and now the first term can be evaluated.

The Tefkat engine implements term ordering at run-time, as Tefkat’s support for late binding and reflection prevents a complete static analysis. We considered including a partial static analysis but the cost of implementing the analysis outweighed the benefits.

5.5 Type constraints for variables

Most transformation rules commence by selecting all instances of particular source types and then constraining them, e.g.

\[
\text{FORALL Class C, Attribute A WHERE C.attr = A}
\]
Implemented naively, this creates $O(M^N)$ branches in the evaluation tree (where $M$ is the number of Class instances and $N$ is the number of Attribute instances), most of which will subsequently fail the test $\text{C}.\text{attr} = \text{A}$. Creating a branch for each instance of a source type is expensive and should be delayed as long as possible (as many might never be needed). Instead of creating these branches, we create a single branch and tag the variable with a type constraint (i.e. C is a Class). Evaluation of a term such as $C.\text{attr} = \text{A}$ discovers that variable C, although still unbound, is sufficiently constrained that binding of variable C is possible and evaluation of this term proceeds by binding values to variable A, provided the values satisfy A’s type constraint. This creates only $O(M)$ branches, since Attribute instances are navigated to rather than retrieving all $N$ instances for each Class instance.

This implementation technique prevents combinatorial explosion in the evaluation tree and significantly improves performance without requiring any changes to the rules themselves.

6. Tool support

The Tefkat engine is available in two forms: as a standalone engine that can be run from a command-line or via a public Java API, and as a plug-in to the Eclipse Integrated Development Environment. The Eclipse plug-in includes an integrated editor for rich feedback to the transformation programmer, a source-level debugger, and integrates with the Eclipse build process to enable automatic re-execution of transformations when sources or the transformation itself have changed.

The integrated editor does syntax colouring, provides in-line warnings about variables that only appear once in a rule (unless they begin with an underscore; a logic-programming convention for indicating that it is a anonymous variable), and displays a hierarchical view of the transformation that groups rules into strata and thus reveals the implicit dependencies between rules.

The source-level debugger allows stepping the engine through the rule evaluation process, and display of variable bindings (including interactive drill-down into Objects). A built-in function, println, is also implemented to enable quick-and-dirty debugging, but the resulting output may not reflect the programmer’s intuition since it is subject to the possibility of term and rule re-ordering and caching.

Most transformations need to perform some string manipulation. Tefkat offers a large library of string processing functions, by using plug-ins to invoke Java string-processing methods (a significant re-use benefit). The plug-in mechanism can also be used by the transformation programmer to call other Java methods; it is the programmer’s responsibility to call only side-effect-free methods. This provides an integration point for calling existing algorithms written in an imperative language. For example in mathematical, engineering, bio-science and many other domains there are large libraries that encode domain-specific algorithms.

7. Future Work

While we feel that the current version of Tefkat (v2.1.0.1) adequately demonstrates the viability of the declarative approach to QVT, nonetheless there is scope for further enhancement.  

7.1 Future design work

The concrete syntax has evolved incrementally, often in response to user feedback. However, as a result, the overall syntax has become irregular and cumbersome in places and needs tidying up. Although we made an original design decision to be somewhat verbose in our concrete syntax to assist programmers unfamiliar with the declarative style, as transformations become larger and more complex, our user community is starting to call for more compact syntactic forms. That is, the verbosity assists the beginning user, but frustrates the more experienced one. We are now considering the design of more compact syntactic forms that can be used alongside the current SQL-like syntax.

In particular, we are considering a tuple style as an alternative for LINKS/LINKING for tracking classes, and also a functional style for PATTERN definitions with explicit input and output variables.

It is often suggested that transformations might be better expressed in a graphical form. While we are interested in experimenting, we see a graphical form as a useful alternative view of a transformation rather than a complete alternative to a textual notation. Specifically, while the high-level nature of the relationships between sources and targets can be depicted graphically, the “devil in the detail” of transformations, often elaborate string manipulation, is difficult to present graphically.

Currently Tefkat offers only one approach to specifying ordering, that is through the specification of partial orders involving two elements that is subsequently resolved into a total order. While this is an excellent general-purpose approach, it commonly results in an $O(N^3)$ evaluation of the ordering rule and an $O(N^3)$ calculation of the total order. While we haven’t yet observed this to be a practical problem, as $N$ is usually small, nonetheless particular transformations could exhibit pathological behaviour. We would like to introduce alternative ways to specify ordering, for example, sort-by terms or explicit (logical) indices.

As Tefkat is being increasingly used for large and complex transformations we now believe that we need to introduce mechanisms for grouping of rules and whole-of-transformation importing and re-use. This may involve the ability to explicitly specify dependencies between groups of rules.

There is also a need for black-box transformation re-use. This is where the inputs and outputs, including trace information, of a number of transformations are plugged together to produce a composite transformation. Our early work with black-box re-use has highlighted the need for special treatment of trace information (from an earlier transformation) that is supplied as input to this transformation. Without this support, treating this information as a conventional “source” produces transformations that are very hard to read.

7.2 Future implementation work

We have also identified some areas in which the Tefkat engine and tool support could be improved.

The current source-level debugger exposes users directly to term reordering and caching, concepts that are otherwise invisible to the user. We would like to explore more declarative approaches to debugging based on the user’s perception of different failure semantics of different terms in their rules. We find that users
expect that some terms can fail because there may be nothing to match, whereas other terms are seen as more computational and hence are expected to always succeed. Considering the example in Figure 1, C.is_persistent = true would be expected to sometimes fail, but not C.name = N. If the engine were aware of this distinction, then a failure of an expected-to-succeed term should be brought to the user’s attention since this may be the original source or first indication of a failure that may not otherwise manifest itself until much later in the evaluation.

Currently term reordering is done by inspecting terms in isolation. If terms were examined in groups, then a more efficient evaluation order may be chosen. For example, a single-valued property should be queried before a multi-valued one to minimize branching in the evaluation tree.

8. Conclusions
In the preceding sections we have demonstrated that a declarative approach to QVT is possible. Tefkat embodies the benefits of the declarative approach while avoiding the perceived disadvantages. This happy outcome reflects both a long programme of research and development in model transformation engines and the feedback of our user community. In particular we must thank our users for feedback that resulted in:

- extensive string manipulation support,
- target-side templates,
- object literal syntax,
- in-line definition of tracking classes
- abstract rules,
- println as a poor-man’s debugger
- warnings about single-use variables and the underscore syntax conventions,
- the virtual Any type.

Intriguingly, while Tefkat adopts a strict approach to declarative transformations, the experiences and lessons are strongly aligned with those reported for ATL [7] which additionally supports rules with imperative aspects. Further work is required to compare the two languages and understand how they each solve/avoid various semantic and implementation challenges such as recursion through negation.

The Tefkat Open Source project is available from SourceForge (http://tefkat.sourceforge.net/) under the Lesser GNU Public Licence (LGPL). The discussions of the Tefkat user community can be found at http://groups.google.com.au/group/Tefkat

9. Acknowledgments
We wish to thank all members of the Tefkat user community with particular thanks to Keith Duddy, David Hearnden, Alejandro Metke, and Jim Steel.

10. References
17. W3C “XSL Transformations (XSLT) Version 1.0”, W3C Recommendation, 16 November 1999