The MT model transformation language

Laurence Tratt
Department of Computer Science, King's College London, Strand, London, WC2R 2LS, U.K.
laurie@tratt.net

ABSTRACT

Model transformations are recognised as a vital part of Model Driven Development, but current approaches are often simplistic, with few distinguishing features, and frequently lack an implementation. The practical difficulties of implementing an approach inhibit experimentation within the paradigm. In this paper, I present the MT model transformation language which was implemented as a low-cost DSL in the Converge programming language. Although MT shares several aspects in common with other model transformation languages, an ability to rapidly experiment with the implementation has led MT to contain a number of new features, insights and differences from other approaches.

1. INTRODUCTION

As the software development community has increasingly embraced the use of models in its development, the need for model transformations has increased, particularly in the context of MDA [1, 8]. A simple definition of a model transformation is that it is a program which mutates one model into another; in other words, something akin to a programming language compiler. Of course, if this simple description accurately described model transformations, then General Purpose Languages (GPL's) would suffice to express model transformations. In practise, model transformations present a number of problems which imply that dedicated approaches are required [13].

In recent times, many different model transformation approaches have been proposed (see e.g. [7, 4] for overviews of different approaches). However I believe only a relatively small part of the solution space has hitherto been explored. It is my contention that the difficulty of implementing model transformation approaches is one of the chief reasons for the relative simplicity of most current model transformation approaches. Only a small proportion of proposed approaches appear to have an implementation, with some of those being too limited to perform any meaningful task.

The purpose of this paper is to present an overview of the MT model transformation language, outlining its basic features, along with some of the additional features it possesses over existing approaches such as the QVT-Partners approach [11], and concluding with a non-trivial example of its use. MT is implemented as a Domain Specific Language (DSL) within the Converge programming language [12]. Converge is a novel Python-derived programming language whose syntax can be extended, allowing DSLs to be directly embedded within it. Some of MT's differences from other approaches are a side-effect of implementing MT as a Converge DSL; some are the result of experimentation with a concrete, but malleable, implementation. Space constraints mean that it is left to an extended technical report to detail MT's complete implementation [14]. Due partly to its implementation as a Converge DSL, MT is novel due to its unique interspersing of declarative and imperative aspects of model transformations, as well as for some of its features which are novel in a model transformation context.

SAC'06 April 23-27, 2006, Dijon, France

2. OVERVIEW

MT is in many senses a derivative of the QVT-Partners approach [11]. The QVT-Partners approach is interesting for its use of patterns – the modelling equivalent of textual regular expressions – which allow the concise expression of constraints over models. However the QVT-Partners patterns are weak in expressive power, can only match against a fixed number of elements, contain flaws in their scoping rules, and make use of a convoluted imperative language for rule bodies. See [14] for more details.

The MT language is a new unidirectional stateless model transformation language, implemented as a DSL within Converge. MT defines a natural embedding of model transformations within Converge, using declarative patterns to match against model elements in a terse but powerful way, whilst allowing normal imperative Converge code to be embedded within rules. Because MT is implemented as a DSL within Converge, it has existed as a concrete implementation from shortly after its original design was sketched out. This has allowed practical experience with the approach to be quickly fed back into the implementation. Rapid experimentation with the implementation has led MT to contain a number of insights and distinct differences from other approaches, such as a more sophisticated pattern language and suitable ways to visualize model transformations.

MT has a sister DSL TM, which allows typed modelling languages to be easily expressed. This is detailed in more detail in [14]. For the purposes of this paper, it is sufficient to know that TM allows UML style modelling languages,

and their instances, to be expressed.

2.1 Basic details

An MT transformation has a name and consists of one or more rules, the ordering of which is significant. Rules are effectively functions which define a fixed number of parameters and which either succeed or fail depending on whether the rule matches against given arguments. If a rule matches successfully, one or more target elements are produced and it is said to have *executed*; if it fails to match successfully, the rule fails and no elements are produced. Rules are comprised of: a source matching clause containing one or more source patterns; an optional when clause on the source matching clause; a target producing clause consisting of one or more expressions; and an optional where clause for the target production clause.

An MT transformation takes in one or more source elements, which are referred to as the *root set* of source elements. The transformation then attempts to transform each element in the root set of source elements using one of the transformations rules, which are tried in the order they are defined. If a given element does not cause any rule to execute then an exception is raised and the transformation is aborted.

The general form of an MT transformation is as follows:

```
import MT.MT

$<MT.mt>:
  transformation transformation name

  rule rule name:
    srcp:
      pattern_1 ... pattern_n

    src_when:
      expr

    tgtp:
      expr_1 ... expr_n

    tgt_where:
      expr_1 ... expr_n
```

The `import` statement is an expression in the base Converge language and imports a module. *DSL blocks* are introduced by `$<...>` – in this example an MT model transformation DSL block. Since Converge is an indentation based language, all code indented from the `$<MT.mt>` line is part of the DSL block; all code outside of the indented block is normal Converge code. As this example shows, a Converge DSL can conform to an arbitrary grammar. A DSL block is translated into a Converge abstract syntax tree using Converge's compile-time meta-programming facilities. As will be seen later, arbitrary Converge code can be embedded inside the DSL block itself. See [12, 14] for more details on these mechanisms.

The `srcp` and `srcp_when` clauses are collectively said to form the *source clauses*; similarly the `tgtp` and `tgtp_when` clauses are collectively said to form the *target clauses*. Transformation rules contain normal Converge code in expressions; such expressions can reference variables outside of the model transformation DSL fragment. Users can thus call arbitrary Converge code, allowing them a means to extend the model transformation approach as necessary.

2.2 Matching source elements with patterns

A pattern in a `srcp` clause is analogous to an parameter in a function. In fact, pattern matching in MT is rather like an extended version of pattern matching in functional languages such as Haskell [9]. Patterns match against arguments passed to a rule, binding successful matches to variables. A *variable binding* in MT is a variable name surrounded by angled brackets '`<`' and '`>`' to distinguish it from a normal Converge variable reference.

The matching algorithm used by MT is intentionally simple. Each pattern in a `srcp` clause in turn attempts to match against the top-level source elements passed in the appropriate argument. Each time a pattern matches it produces variable bindings which are available to all subsequent patterns. If a pattern fails to match, control backtracks to previous patterns (in the order of most recently visited), which will then attempt to generate new matches given the variable bindings and arguments available to it. The generation of an alternative match causes new variable bindings to be produced, which allows the rule to attempt another match of later patterns. The `src_when` clause, if it exists, must be a single Converge expression and is evaluated once all patterns have been matched successfully; it is essentially a guard over patterns. If it fails, patterns are requested to generate new matches exactly as in the failure of a pattern to match. If all patterns, and the `src_when` clause match successfully, then the rule executes

The order that patterns are defined in the `srcp` clause is significant, for two separate reasons. Most obviously it is necessary to ensure that users sequence variable bindings and references to the bound variables correctly. However there is a second reason that, whilst less obvious, is critical to the performance of larger transformations. Making the order of patterns significant allows users to make use of their domain knowledge to order them in an efficient way. Consider a rule which has two independent patterns x and y where x tends to match against many source elements, but y against few. Placing x first in the `srcp` clause means that when y fails x will try to produce more values; if x can produce multiple matches, y may be executed many times unnecessarily. If y is placed first in the `srcp` clause then if it fails to match against its input the rule fails without ever trying to match x. Sensible ordering of patterns in this way can lead to a significant boost in performance as unnecessary matches are not evaluated.

2.3 Pattern language

MT's pattern language is a super-set of that found in the QVT-Partners approach. MT defines a number of *pattern expressions*: model element patterns, set patterns, variable bindings, and normal Converge expressions. Model element patterns are of the form `(Class, <self_var>)[slot name == pattern]`. A model element pattern matches against a model element of type `Class`, and then checks each *slot comparison* `slot name` against a pattern `pattern`. If the type check, or any of the slot comparisons, fails then the entire model element pattern fails. In general, any of the standard Converge comparison operators (e.g. `==`, `>=` etc.) can be used in slot comparisons, and the same slot name may be involved in multiple comparisons in any given model element expression. If the type of the model element pattern, or any slot comparisons fail, then the model element pattern itself fails. Set patterns are directly analogous to those found in functional languages such as Haskell. Variable bindings `<v>` match against any model element, binding the element to v; patterns subsequent to the binding may use the variable v

in the normal fashion. Converge expressions, when used as patterns, match only against a model element which compares equal to the evaluated Converge expression. If a model element expression successfully matches against a model element, then the model element is bound to the optional *self variable* `self_var`.

As a trivial example of a model element pattern, assuming an appropriate meta-model, the following example will match successfully against a `Dog` model element whose owner is not Fred, binding the `Dog` element to the variable `d` and its name to `n`:

```
(Dog, <d>)[name == <n>, owner == (Person)[name != "Fred"]]
```

As a point of comparison, this example would necessitate an OCL constraint in a `when` clause in the QVT-Partners approach since that approach does not possess slot comparisons other than simple equality.

2.4 Producing target elements

When an MT rule executes it produces one or more target elements. As this implies, MT is distinct from e.g. graph transformation approaches which take an input and gradually mutate it into the output. MT takes an input and produces a fresh output; at no point is the input model altered.

If a rule executes but fails to produce any elements, an exception is raised. The number of elements produced is determined by the number of expressions in the `tgtp` clause. Each expression is a normal Converge expression, but with an important addition. The MT DSL admits *model element expressions* by extending Converge's builtin grammar. Model element expressions differ from model element patterns both conceptually and syntactically. Conceptually a model element expression is an imperative, creational action; to reinforce this notion, *slot assignments* in a model element expression use the normal Converge assignment operator `:=`.

Expressions in `tgtp` have an optional `for` suffix which allows a single expression to generate multiple values. If one ignores the obvious syntactic difference of the relative location of the keyword, the `for` suffix works largely as a standard `for` construct, taking a single expression and continuously producing a model element for each iteration of the loop. Variables defined in the `for` suffix are scoped only over the single expression in the `tgtp` clause that it suffixes.

The `tgt_where` clause, if it exists, is a sequence of Converge expressions which are executed before the `tgtp` clause. Variables in the `tgt_where` clause are automatically scoped over the `tgtp` clause. Unlike the `src_when` clause, there is no notion of success or failure with the `tgt_where` clause, which is simply a helper function for the `tgtp` clause. Note that expressions in the `tgt_where` clause can contain model element expressions.

2.5 Pattern multiplicities

One problem with approaches such as the QVT-Partners approach is that model element patterns can only match against a fixed number of elements. Some very simple transformations naturally consist only of rules which match against a fixed number of elements in the source model. However, many, if not most, non-trivial transformations contain rules which need to match against an arbitrary number of source elements. Expressing such transformations in the QVT-Partners approach, and indeed many other model transformation approaches, requires cumbersome work arounds [14].

To solve this problem, MT adapts the concept of multiplicities found in many textual regular expression languages. Each source pattern in MT can optionally be given a *multiplicity* and an associated variable binding. Multiplicities specify how often a given source pattern can, or must, match against its source elements. Multiplicities are a constraint on the universe of model elements passed in the parameter corresponding to the patterns position in the `srcp` clause. The following example of a pattern multiplicity will match zero or more dogs whose owner is Fred, assigning the result of the match to the `dogs` variable:

```
(Dog, <d>)[owner == (Person)[name == "Fred"]] : * <dogs>
```

The syntax for multiplicities is inspired by Perl-esque regular expression languages. Amongst the multiplicities, and possible qualifiers, defined in MT are the following:

m	Must match exactly m source elements.
*	Will match against zero or more source elements.
* !	Must match against every source element.
m .. *n*	Must match no less than m, and no more than n source elements.
m .. * ?	Will match against the minimum number of source elements once `m` elements have been matched.

As with Perl-esque textual regular expressions, multiplicities default to 'greedy' matching — that is, they will match their pattern against the maximum number of elements that causes the multiplicity to be satisfied. When backtracking in a `srcp` clause calls upon a multiplicity to provide alternative matches, it then returns matches of lesser lengths. The concept of greedy and non-greedy matching is simple in the case of textual regular expressions since text is an inherently ordered data type; the length of a match is calculated by determining how many characters past a fixed starting point a match extends. In contrast to this, model elements have no order with respect to one another, and MT has to take a very different approach to the concepts of greedy and non-greedy matches. MT defines the length of a multiplicities' match as the number of times the multiplicity matched; however since model elements are not ordered, this does not present an obvious way of returning successively smaller matches. In order to resolve this problem in the case of greedy matching, MT creates the powerset of matches, and iterates over it, successively returning sets with smaller number of elements when called upon to do so. Note that whilst MT guarantees that with greedy matching $|match_n| \geq |match_{n+1}|$, it makes no guarantees about the order that sets of equal size in the powerset will be returned.

The `?` qualifier reverses the default greedy matching behaviour, attempting to match the minimum number of elements that causes the multiplicity to be satisfied, successively returning sets of greater size from the powerset when called upon to do so. The `!` qualifier is the 'complete' qualifier which ensures that the pattern matches successfully against every model element passed in the patterns appropriate argument. Whilst the `?` qualifier, in a slightly different form, is standard in most textual regular expression languages, the `!` qualifier is specific to MT.

2.5.1 Variable bindings in the presence of multiplicities

Variable bindings in patterns suffixed by multiplicities need to be treated differently from variables in bare patterns. When a multiplicity is satisfied, its associated variable binding is assigned a list of dictionaries[1]. Each dictionary contains the variable bindings from a particular match of the pattern. The need for different treatment of variable bindings inside and outside multiplicities is most easily shown by examining what would happen if they were treated identically. Consider the following incorrect MT code:

```
(Dog)[owner == (Person)[name = <n>]] : * <ds>
(Person, <p>)[name == n]
```

A first glance may suggest that when the rule these patterns are a part of runs, `p` will be set to the person who owns the dog. However, the example is nonsensical since `n` has no single value. Indeed `n` may have no value at all, since it will be bound to zero or more owners' names as the multiplicity attempts to match the model pattern as many times as possible. As this example shows, `n` has no meaning outside of the multiplicity it is bound in; however it clearly has a meaning in the context of the multiplicity.

In order to resolve this quandary, MT takes a two stage approach. Within multiplicities, local variable bindings are accessed as normal. At the end of each successful match, MT creates a dictionary relating variable binding names to their bound values. The list of these values is then assigned to the variable binding associated with the multiplicity. Thus the variable bindings for each individual match can be accessed. To illustrate this, I reuse the original multiplicities example:

```
(Dog)[owner == (Class)[name = <n>]] : * <ds>
```

Printing the `ds` variable would lead to output along the following lines:

```
[Dict{"n" : "Fred"}, Dict{"n" : "Barney"}]
```

The `MT` module provides a convenience function `mult_extract(bindings, name)` (which, unlike the `transform` of section 3 is not specific to any particular transformation) which iterates over a list of dictionaries, as generated by a multiplicity, and extracts the particular binding `name` from each dictionary, returning a list. A standard idiom when using multiplicities in MT is to use the `mult_extract` function with a model element pattern with a self variable binding.

2.6 Tracing information

MT transformations hold a record of tracing information, which is automatically created as transformation rules are executed. Each rule executed adds a new trace. Each trace is a tuple of the form `[[`*source elements*`], [`*target elements*`]]`. All elements created by model element expressions are automatically stored in the tuple. However by default, only elements matched by non-nested model element patterns are recorded in the tracing information; this means that the source elements that are stored in the tracing information do not necessarily constitute the entire universe of elements passed via parameters to the transformation. Non-nested model element patterns are defined to be those which are not nested within another model element pattern. For example in the following model element pattern, tracing information will be created only from instances of the `Dog` model class:

```
(Dog, <d>)[name == <n>, owner == (Person)[name != "Fred"]]
```

It may seem somewhat arbitrary to try to minimise the source elements used in tracing information since MT maximises the target elements used. The reason for minimising the source elements used is due to a simple observation: individual source elements are often matched in more than one rule execution. This then causes some source elements to be the source for large numbers of traces which can obscure the result of the transformation. Empirical observations of MT transformations suggest that when model elements are matched via nested model element patterns, they are also matched by a non-nested model element pattern during a separate rule execution. In the case of target elements, a different challenge emerges. Rather than trying to create an 'optimum' amount of traces one wishes to ensure that, as far as is practical, every target element has at least one trace associated with it. Since target element expressions are inherently localised to individual rule executions it is highly unusual for an element created by such an expression to be the target of more than one trace. Thus it is important to ensure that nested target element expressions have traces associated with them. [14] shows how nested model element patterns in MT can be made to contribute towards tracing information if desired.

2.6.1 Augmenting or overriding the standard mechanism

Whilst the standard tracing creation mechanism performs well in many cases, users may wish to augment, or override, the default tracing information created. Users may wish to add extra tracing information to emphasise certain relationships within a transformation, or to remove certain tracing information that unnecessarily clutters the transformation visualization. Although not detailed due to space constraints, MT provides a simple capability for augmenting, or overriding, the default tracing information created by the standard mechanism via optional `tracing_add` and `tracing_override` clauses in a transformation rule.

3. EXAMPLE

The example I use in this paper is a variant of the standard class to relational model transformation as found in [11], and which should be consulted for a more complete description of the transformation. The 'Simple UML' meta-model is shown in figure 1, and the relational database meta-model in figure 2. In essence, classes which have the `is_persistent` attribute set to true will be transformed to tables; references to such classes (via attribute types or associations) will result in the classes primary key attributes being to converted to columns used as a foreign key. Classes which do not have the `is_persistent` attribute set to true will not be transformed into tables, and will be 'flattened' when a persistent class is transformed into a table; the attributes of non-persistent classes are prefixed with the name of the class when so flattened. When transforming a class, all associations for which the class is a `src` must be considered. Attributes can be marked as being part of a classes primary key by having the `is_primary` attribute set to true. Note that associations play no part in determining a classes primary key.

This example, whilst relatively simple, is interesting because of considerations such as the following:

[1]Dictionary is Converge's name for the datatype sometimes known as a hash table or associative array.

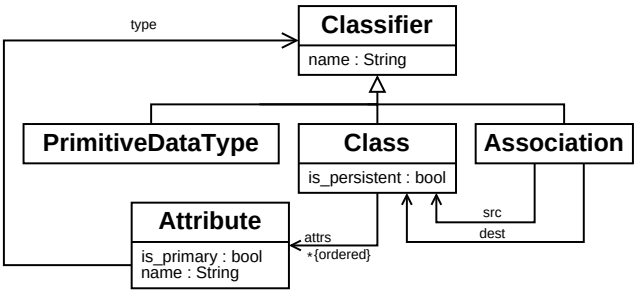

Figure 1: Extended 'Simple UML' meta-model.

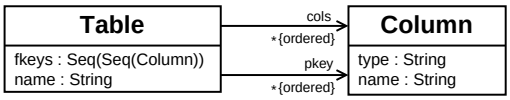

Figure 2: Extended relational database meta-model.

- Classes can not be transformed in isolation – all associations for which a class is the source must be considered in order that the table that results from a class contains all necessary columns.
- Classes which are marked as persistent must be transformed substantially different from those not marked as persistent.
- Foreign keys and primary keys both reference columns; it is important that the column model elements pointed to by a table are the appropriate model elements, and not duplicates.

Noting that booleans in MT are represented by `0` (false) and `1` (true), the MT version of this example is as follows:

```
$<MT.mt>:
  transformation Classes_To_Tables

  rule Persistent_Class_To_Table:
    srcp:
      (Class, <c>)[name == <n>, attrs == <attrs>, \
        is_persistent == 1]
      (Association, <assoc>)[src == c] : * <assocs>

    tgtp:
      (Table)[name := n, cols := cols, pkey := pkeys, \
        fkeys := fkeys]

    tgt_where:
      cols := []
      pkeys := []
      fkeys := []
      for aa := (attrs + MT.mult_extract(assocs, "assoc")) \
        .iterate():
        a_cols, a_pkeys, a_fkeys := self.transform([""], [aa])
        cols.extend(a_cols)
        pkeys.extend(a_pkeys)
        fkeys.extend(a_fkeys)

  rule Primary_Primitive_Type_Attribute_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Attribute)[name == <attr_name>, type == \
        (PrimitiveDataType)[name == <type_name>], \
        is_primary == 1]

    tgtp:
      [col]
      [col]
      []

    tgt_where:
      col := (Column)[name := concat_name(prefix, \
        attr_name), type := type_name]

  rule Non_Primary_Primitive_Type_Attribute_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Attribute)[name == <attr_name>, type == \
        (PrimitiveDataType)[name == <type_name>], \
        is_primary == 0]

    tgtp:
      [(Column)[name := concat_name(prefix, attr_name), \
        type := type_name]]
      []
      []

  rule Persistent_User_Type_Attribute_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Attribute, <attr>)[name == <attr_name>, type == \
        (Class, <class_>)[name == <class_name>, attrs == \
        <attrs>, is_persistent == 1]]

    tgtp:
      cols
      []
      [cols]

    tgt_where:
      cols := []
      for attr := attrs.iterate():
        a_cols, a_pkeys, a_fkeys := self.transform(\
          [concat_name(prefix, attr_name)], [attr])
        cols.extend(a_pkeys)

  rule Non_Persistent_User_Type_Attribute_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Attribute, <attr>)[name == <attr_name>, type == \
        (Class, <class_>)[name == <class_name>, attrs == \
        <attrs>, is_persistent == 0]]

    tgtp:
      cols
      []
      []

    tgt_where:
      cols := []
      for attr := attrs.iterate():
        a_cols, a_pkeys, a_fkeys := self.transform( \
          [concat_name(prefix, attr_name)], [attr])
        cols.extend(a_cols)

  rule Persistent_Association_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Association)[name == <attr_name>, dest == (Class, \
         <class_>)[name == <class_name>, attrs == <attrs>, \
         is_persistent == 1]]

    tgtp:
      cols
      []
      [cols]

    tgt_where:
      cols := []
      for attr := attrs.iterate():
        a_cols, a_pkeys, a_fkeys := self.transform( \
          [concat_name(prefix, attr_name)], [attr])
        cols.extend(a_pkeys)

  rule Association_Non_Persistent_Class_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Association)[name == <attr_name>, dest == (Class, \
        <class_>)[name == <class_name>, attrs == <attrs>, \
        is_persistent == 0]]
      (Association, <assoc>)[src == class_] : * <assocs>

    tgtp:
```

```
      cols
      []
      fkeys

    tgt_where:
      cols := []
      fkeys := []
      for aa := (attrs + MT.mult_extract(assocs, "assoc")).\
        iterate():
        a_cols, a_pkeys, a_fkeys := self.transform( \
          [concat_name(prefix, attr_name)], [aa])
        cols.extend(a_cols)

rule Default:
  srcp:
    (MObject)[]

  tgtp:
    null
```

In order to run this transformation, a list of top-level elements (classes and associations) should be passed to it. There is no requirement to designate one particular class as being the 'start' class for the transformation. The output of the transformation will consist of a number of tables.

The overall structure of this transformation is hopefully relatively straight forward even if some of the finer details are not. The `Persistent_Class_To_Table` rule ensures that each class marked as being persistent in the source model is transformed into a table in the target model. It takes a persistent class, and finds all of the associations for which the class is a source; it then iterates over the union of the classes' attributes and associations for which it is a source, transforming them into columns. All of the other rules take in a string prefix (representing the column prefix being constructed as the transformation drills into user types), and an attribute or association (and, in the case of the `Association_Non_Persistent_Class_To_Columns` rule, an additional set of associations) and produces three things: a list of normal table columns; a list of primary key columns; a list of foreign key columns. The final rule in the transformation `Default` is a 'catch all' rule that takes in model elements from the root set which not matched by other rules – non-persistent classes and associations – and transforms them into the `null` object; this causes MT to discard the result of the transformation rule, and not create any tracing information. The `Default` rule is necessary to ensure that such elements in the root set of source elements do not cause the transformation to raise a `Can not transform` exception.

Two features in this transformation need extra explanation in the context of this paper. The `transform` function used throughout the transformation is present in every MT transformation. It takes an element(s) in, and successively tries every transformation rule in the transformation using the arguments passed to it, attempting to find one which executes given the element(s) as input. If no rule executes, the `transform` function raises an error. The `transform` function is used internally by MT to transform each element in the root set but, as in this example, may be called at will by the user.

The second feature that requires explanation leads on from the first, but is less obvious to the casual reader. Many of the rules have more patterns than there are arguments passed to the `transform` function. The `Association_Non-_Persistent_Class_To_Columns` rule, for example, defines three patterns but the `transform` function is never called with more than two arguments – it would thus seem impossible for this rule to ever execute. However, MT defines that when a rule is passed fewer arguments than it has parameters, the root set of source elements is substituted for each missing argument. This is effectively an escape mechanism allowing rules access to the complete source graph. This mechanism is vital for ensuring that transformations such as this are not complicated by the need to pass the root set of source elements to every rule execution.

4. AN EXECUTION

Figure 3 shows a complete run of the example transformation on a simple input, automatically visualized as a hybrid object diagram. Input elements are shown as rounded boxes, with links between input elements being shown with dashed lines. Output elements are shown in non-rounded boxes, with solid links. The dotted links are tracing information and are explained in section 4.1. An execution of this transformation with a significantly larger input model can be found in [14], which also documents MT's other visualization techniques and options.

4.1 Visualizing tracing information

Visualizing tracing information is an interesting challenge, and one that has hitherto received scant attention in the context of model transformations. Work on trace visualization in areas such as object orientated systems (e.g. [2]) is of little use in the context of model transformations. Egyed motivates the use of tracing information in the context of modelling, but explains neither how to generate or visualize such information [6]. MT and TM cooperate together to present a simple visualization of tracing information that also allows users to build up a detailed picture of how the transformation executed.

In figure 3, the dotted lines between source and target elements represent the individual traces between source and target elements. To avoid cluttering, the visualization of a trace is always from a single source element to a single target element. Each trace has a name of the form tn where n is an integer starting from 1. The integer values reflect the traces position in the execution sequence; trace numbers can be compared to one another to determine whether a rule execution happened earlier or later in the execution sequence. Trace names can be looked up in the 'Tracing' table at the top right of figure. The tracing table contains the name of each rule which was executed at least once during the transformation. Against each rule name are the names of traces; each trace name represents an execution of that rule. Note that a single rule execution can create more than one trace; however each trace created in a single execution will share the same name.

Although the visualization of tracing information may seem simple, it allows one to infer a great deal of useful information about the execution of a transformation. This information is useful both for analysis and debugging of a transformation. At a simple level, one can use the names of tracing information to determine which rule consumed which source elements and produced which target elements. For example a trace marked 't1' is a result of an execution of the `Persistent_Class_To_Table` rule. One can also deduce from this traces name that it was the result of the first rule execution in the system. Equally since two traces share the name 't1', one can determine that during a single execution the `Persistent_Class_To_Table` rule produced two

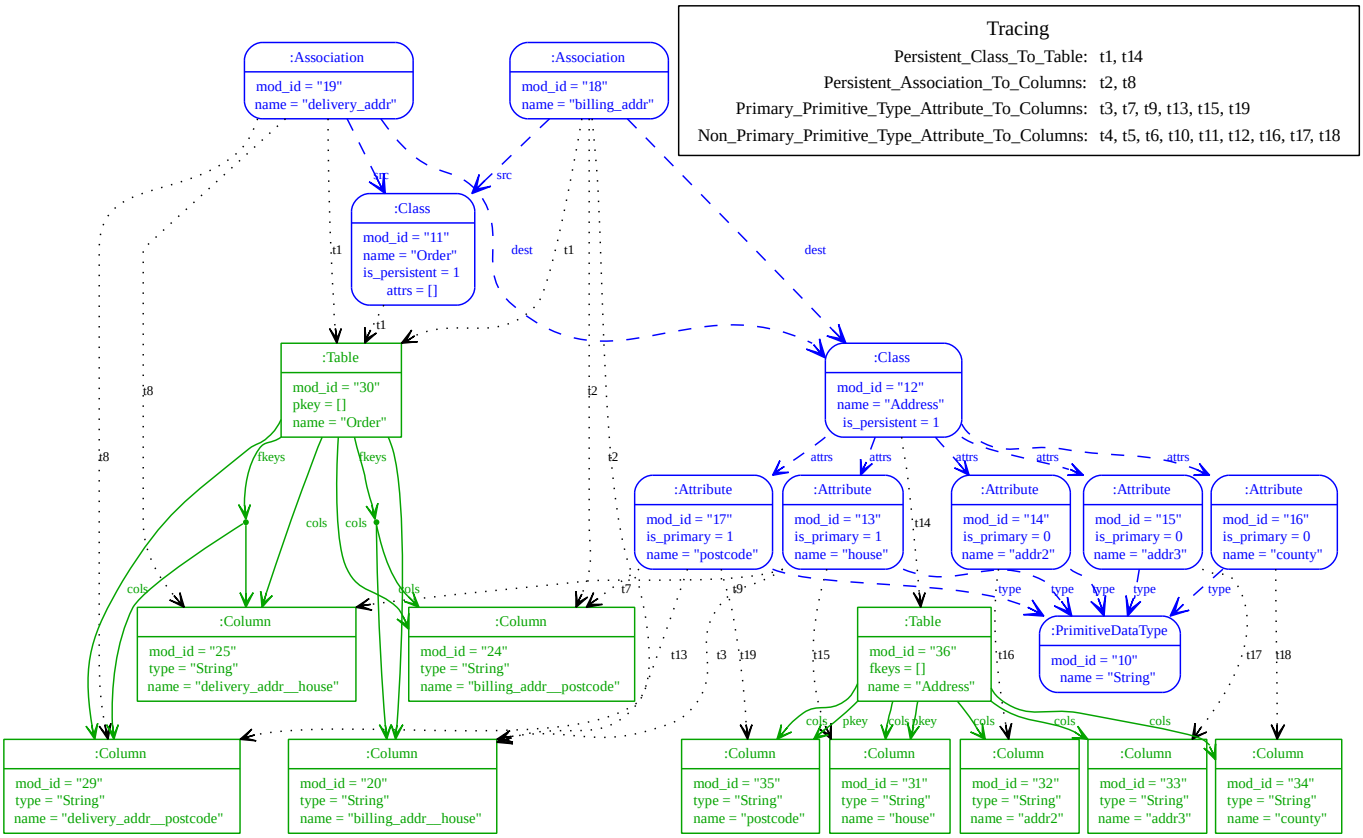

Figure 3: An example execution of the extended transformation.

elements.

4.2 Pruning the target model

One thing not immediately obvious from viewing figure 3 is that, unlike the majority of model transformation approaches, MT does not force the final target model to be a union of the model elements produced in every rule execution. In fact, if one were to take the union of model elements produced by every rule execution, the target model would contain many superfluous model elements. The reason for this can be seen by examining a rule such as `Persistent_-Association_To_Columns`. This rule calls the `transform` function but then effectively discards some of the model elements produced by this call (the rule in question cares only about primary key columns, and ignores any non-primary key columns that may have been produced). Knowing that, as an implementation detail, TM assigns each new model element a unique and monotonically increasing identifier, one can see from figure 3, that some elements have been discarded, due to the non-contiguous model identifiers in target model elements. The lowest `mod_id` for a target model element is 20 and the highest 36, but only 11 elements with identifiers in that range are present in the output – the missing 5 identifiers are evidence that elements were produced by a rule execution, but subsequently discarded.

To determine the final target model, MT's uses the model elements that resulted from transforming each element in the root set. It uses these elements as the root nodes in a simple graph walking scheme. Only target model elements which are reachable from these elements are considered to be part of the eventual target model. Although the example in this paper does not demonstrate it, this scheme does allow the eventual target model to consist of unconnected subgraphs.

5. RELATED WORK

As with the majority of existing model transformation systems, MT is a unidirectional stateless model transformation system (in the language of [13]). MT's most obvious ancestor is the QVT-Partners approach [11] which pioneered the use of patterns in model transformations and which is a core part of the forthcoming QVT standard [10]; the commercial XMap language [3] is also similar. MT takes the base QVT-Partners pattern language and enriches it with features such as pattern multiplicities and variable slot comparisons. Converge's approaches to generating and visualizing tracing information, and its pruning of the output model have no precedent in the QVT-Partners approach. A significant difference from the QVT-Partners approach is in MT's imperative aspects. Due to its implementation as a Converge DSL, MT can embed normal Converge code within it. This contrasts sharply with the QVT-Partners approach which is forced to define an OCL variant with imperative features in order to have a usable language. The forthcoming QVT standard also separately includes wholly declarative and wholly imperative model transformations; MT can be thought of as treading the ground in between these two extremes, gaining many of the advantages of both, whilst avoiding many of their respective pitfalls. Section 6

describes some early work which extends MT in directions beyond that of the forthcoming QVT standard.

Perhaps surprisingly, given the seeming simplicity of the task, one of MT's most distinctive features is its automatic creation of tracing information. Most approaches neglect this problem; the few that tackle it, such as the DSTC approach [5], require the user to manually specify the tracing information to be created. Using patterns defined by the user to automatically derive tracing information has not, to the best of my knowledge, been used by any other system. MT distinguishes itself further by its simple, but effective, technique for reducing superfluous tracing information.

6. FUTURE WORK

Although I believe that MT is currently one of the most advanced model transformation languages available, the relative immaturity of the area means that no new approach can claim to present a definitive solution. Perhaps the most pressing question for every model transformation approach, including MT, regards scalability. Although MT has been used to express transformations of the order of magnitude of the low tens of rules, it is clear that in order to make larger transformations feasible, new techniques for structuring and combining rules will be required. For example, currently all the rules in a MT transformation exist in a single namespace; there is no notion of 'transformation modules'. Tackling this problem is perhaps the most important step in maturing model transformations as an area.

MT has been used as the basis for a change propagating model transformation approach, which raise a number of new challenges above the stateless model transformations MT is capable of performing. The change propagating approach will be documented in a follow up paper.

7. CONCLUSIONS

In this paper I presented the MT model transformation language. After detailing MT's basic features, I presented features novel in the field of model transformations, such as its visualization of transformation executions, its strategies for automatically generating tracing information, its definition of pattern multiplicities, and its approach to producing the final target model. I then made use of these novel features in a non-trivial example.

An extended version of this paper can be found in the technical report [14]. MT can be found as part of the Converge programming language, freely available under a MIT / BSD-style licence from `http://convergepl.org/`.

This research was funded by a grant from Tata Consultancy Services.

8. REFERENCES

[1] J. Bézivin and S. Gérard. A preliminary identification of MDA components. In *Generative Techniques in the context of Model Driven Architecture*, Nov 2002.

[2] H.-D. Böcker and J. Herczeg. What tracers are made of. In *Proc. ECOOP '90*, pages 89–99, 1990.

[3] T. Clark, A. Evans, P. Sammut, and J. Willans. Applied metamodelling: A foundation for language driven development, September 2004. Available from `http://www.xactium.com/` Accessed Sep 22 2004.

[4] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In J. Bettin, G. van Emde Boas, A. Agrawal, E. Willink, and J. Bézivin, editors, *Second Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

[5] DSTC, IBM, and CBOP. MOF query / views / transformations first revised submission, August 2003. OMG document `ad/2003-08-03`.

[6] A. Egyed. A scenario-driven approach to traceability. In *Proc. 23rd International Conference on Software Engineering*, pages 123 – 132, 2001.

[7] T. Gardner, C. Griffin, J. Koehler, and R. Hauser. Query / views / transformations submissions & recommendations towards final standard, August 2003. OMG document `ad/03-08-02`.

[8] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformation: First International Conference, ICGT 2002*, pages 90–105, October 2002.

[9] S. P. Jones. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[10] QVT-Merge second revised submission to QVT RFP, 2003. OMG document `ad/2005-03-02`.

[11] QVT-Partners. First revised submission to QVT RFP, August 2003. OMG document `ad/03-08-08`.

[12] L. Tratt. The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London, February 2005.

[13] L. Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[14] L. Tratt. The MT model transformation language. Technical Report TR-05-02, Department of Computer Science, King's College London, May 2005.

Biography

Laurence Tratt is a Research Fellow at King's College London. His research interests include practical applications of modelling languages, model transformations, domain specific languages. He is the chief designer and implementer of the Converge programming language.