Model transformations in MT

Laurence Tratt[a]

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

Model transformations are recognised as a vital aspect of Model Driven Development, but current approaches cover only a small part of the possible spectrum. In this paper I present the MT model transformation which shows how a QVT-like language can be extended with novel pattern matching constructs, how tracing information can be automatically constructed and visualized, and how the transformed model is pruned of extraneous elements. As MT is implemented as a DSL within the Converge language, this paper also demonstrates how a general purpose language can be embedded in a model transformation language, and how DSL development can aid experimentation and exploration of new parts of the model transformation spectrum.

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 [4,12]. 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 (GPLs) would suffice to express model transformations. In practise, model transformations present a number of problems which imply that dedicated approaches are required [28].

In recent times, many different model transformation approaches have been proposed (see e.g. [11,8] 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.

In this paper I present the MT model transformation language which can be considered as a 'post-QVT' [22] model transformation approach. Compared to other model transformation approaches, MT has novel pattern matching features inspired by textual regular expressions, a practical approach to the automatic creation, customisation, and visualization of tracing information, and the ability to prune the transformed model of extraneous model elements. MT is implemented as a Domain Specific Language (DSL) within the

Converge programming language [27]. 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. A sub-aim of this paper is therefore to demonstrate that model transformation approaches are often good candidates for implementation as low-cost DSLs, and that this can aid experimentation and exploration of the spectrum of possible model transformation approaches.

This paper is an extended version of [30]. In this extended paper, I explore in depth the differences between QVT and MT, explore tracing visualization in greater detail, present a more thorough case for, and explanation of, model pruning, and show how MT supports transformation combinators. A separate technical report contains further details of MT's complete implementation [29], including line-by-line analysis of much of the underlying MT implementation.

The structure of this paper is as follows. In section 2 I examine the QVT-Partners approach [22], identifying issues with the approach. In section 3 I define the MT language in detail, starting with a basic overview, before defining its novel pattern matching features, and approach to creating tracing information. In section 4 I then present a substantial example MT model transformation. In section 5 I show a particular execution of the example model transformation, demonstrating MT's visualization of tracing information and its approach to pruning extraneous elements from the transformed model.

2. Background

MT is in many senses a derivative of the QVT-Partners approach [22]. In this section I therefore briefly overview the QVT-Partners approach, outlining some of its limitations (a comparison of MT with other approaches is presented in section 6). As MT is implemented as a Converge DSL – and since this plays a significant rôle in the rest of this paper – I also present a brief overview of Converge.

2.1. The QVT-Partners approach

In this section, I explain some relevant aspects of the QVT-Partners approach, since the MT language shares several factors in common with the QVT-Partners approach. Whilst the QVT-Partners approach has the concept of 'specification' and 'implementation' transformations, the former are largely irrelevant in the context of this paper and are ignored, as is the diagrammatic syntax for transformations which the approach defines.

The QVT-Partners approach to model transformations is particularly interesting for its use of patterns – the modelling equivalent of textual regular expressions – which allow the concise expression of constraints over models. The QVT-Partners approach identifies three main types of patterns: set, sequence, and object patterns. To ensure consistency with the rest of this paper, I henceforth refer to object patterns as *model element patterns*. Although not explicitly noted as such, variables in patterns are essentially patterns themselves. All types of pattern share in common one thing: given a particular model element, they will either succeed or fail to match against it.

Set and sequence patterns are similar to those used in function parameters in functional programming languages such as Haskell. For example a set pattern `Set{1, 6 | R}` will

match successfully against a set that contains at least two items `1` and `6`; a new set containing all of the original sets items other than `1` and `6` will be bound to `R`. Intuitively, variable names mean 'match anything and bind'; henceforth these will be referred to as *variable bindings*. If the same variable name appears more than once in the same scope, all instances of that variable name must match against equivalent objects (the QVT-Partners approach does not define its own notion of equality, instead inheriting it from the MOF [21]).

Although relatively simple, model element patterns are the backbone of the pattern language. Model element patterns specify the type that matching model elements must conform to, and an optional 'self' variable which will be bound to the element matched against. The model element pattern then specifies a number of a slots and a pattern against which each slot in the model element must match against. The terse power of model element patterns is best demonstrated by example. Consider first the following model element pattern:

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

This pattern will match successfully against a model element which is of type `Dog` and whose owner is Fred. After the match the variable `d` will point to the particular `Dog` element matched, and `n` will contain the dog's name. The equivalent OCL constraint is significantly longer, and inscrutable [29]; patterns allow complex constraints to be tersely expressed, allowing the model transformation writer to concentrate on the salient points of their task unencumbered by unnecessary machinery.

2.1.1. Issues with the approach

In [29] an example encoding of the standard 'class to relational database' model transformation in the QVT-Partners approach is presented; despite the seeming power of patterns, the resultant transformation is longer than its equivalent in a GPL. As this may suggest, the QVT-Partners approach has a number of minor flaws and limitations which hamper practical use. In this subsection I outline, in approximately descending order of importance, three areas which are indicative of where the QVT-Partners falls short of its intended goals:

Inappropriate imperative language The imperative bodies of mappings are written in a so-called 'extended OCL', which is intended to allow users familiar with OCL the chance to reuse that knowledge in an imperative setting. This has an immediate negative effect: extending OCL with imperative constructs means that the often desirable properties OCL had as a purely side-effect free language are lost[1]. Conversely, when it comes to acting as a normal GPL the resulting language is unwieldy since it lacks appropriate constructs for common operations. For example, there is no explicit sequencing mechanism: the imperative body consists of exactly one OCL constraint, and sequencing can only be achieved clumsily via the `let` expression.

Underpowered patterns The pattern language defined in the QVT-Partners approach is novel in the context of model transformations, and potentially very useful. However the pattern language is lacking in significant expressive power. For example

[1]OCL 2.0 is not in fact entirely side-effect free; however the situations in which this property is violated are largely irrelevant in the context of this paper.

within model element patterns it is only possible to check for the equality of slots e.g. it is not possible to use a model element pattern to express that a match against an object should succeed provided a given slot does *not* match a particular value. Furthermore patterns can only match against a fixed number of elements. A model element pattern, for example, can only match successfully against one, and only one, model element. Note that whilst set and sequence patterns can match against sets and sequences of arbitrary lengths, only a fixed number of elements can be explicitly identified within any given set or sequence. There is no general solution to this problem; typically a new mapping needs to be added so that iteration in a `when` clause can control the number of times another mapping is successfully matched.

Scoping rules Since a bare variable name in a slot constitutes a variable binding, the QVT-Partners approach has fragile scoping rules, since it is difficult to distinguish a variable binding from a variable reference; whilst not a problem in some languages such as Prolog, this leads to ambiguities in this case. Consider the simple pattern given earlier in this section; `Dog` is a reference to the `Dog` model class, whereas `d` is a variable binding which will be set to the self value of the object which matches the model element pattern. This has several different consequences [29]. As a simple example, it is impossible to express that a model element pattern should match against a particular element. So if `d` was intended to refer to a specific dog element, that would be ignored and a new local variable binding `d` introduced (which may or may not point to the intended dog model element).

The QVT-Partners approach provides a number of innovations compared to other model transformation approaches, most notably the use of patterns. However in practise the simplistic nature of the approach means that it falls somewhat short in its aim to allow users to express model transformations more easily than in GPLs.

2.2. Converge

Converge is a dynamically typed, imperative, object orientated programming language with compile-time meta-programming facilities. Converge's most obvious ancestor is Python [31] resulting in an indentation based syntax, a similar range and style of datatypes, and general sense of aesthetics. Converge's expression language is heavily based on the goal-directed evaluation style found in Icon [13], where expressions can either *succeed* or *fail*, and limited backtracking can occur within programs.

For the purposes of this paper, compile-time meta-programming can be largely thought of as being equivalent to macros in the LISP (rather than C) sense; more formally, it allows the user of a programming language a mechanism to interact with the compiler to allow the construction of arbitrary program fragments by user code. Converge's compile-time meta-programming facility is inspired by that found in Template Haskell [25], and is detailed in depth in [26]. A simple layer on top of the compile-time meta-programming facility allows DSLs with their own arbitrary syntaxes to be embedded in Converge source code. The advantage of using this approach is that many aspects of DSL development are handled by Converge itself, significantly simplifying development; furthermore the DSL itself can interact arbitrarily with normal Converge code.

3. MT

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 [29]. For the purposes of this paper, it is sufficient to know that TM allows UML-style modelling languages, and their instances, to be expressed in a flat namespace. MT uses TM's modelling languages and instances.

In this section I describe MT, starting first with its basic details; I then explain its novel features and approaches, such as pattern multiplicities, and model pruning.

3.1. 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>:
  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 a normal expression in the Converge language and imports the `MT` 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; note that code preceding `$<MT.mt>` is normal Converge code, as is any code following the DSL block. 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 [27,29] 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; furthermore the expression language used to extend transformations is the same as that used within transformations.

3.2. Matching source elements with patterns

A pattern in a `srcp` clause is analogous to a 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 [15]. Patterns match against arguments passed to a rule, binding successful matches to variables. A *variable binding* `<v>` in MT is a variable name surrounded by angled brackets and causes whatever object is matched by the binding to be assigned to v. This distinguishes it from a normal Converge variable reference (and thus solves the scoping problem identified with the QVT-Partners approach in section 2.1.1). Note that while it is legal (if rarely useful) to bind a variable multiple times (each binding essentially overwrites the previous value), a variable reference x can not precede a variable binding `<x>`.

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.

3.3. 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 were discussed in the previous section. 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 matching `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, which does not possess slot comparisons other than simple equality.

Allowing different types of slot comparison in model element patterns opens up new possibilities. Since MT allows the same slot name to appear in more than one slot comparison, one can test a slot for multiple conditions as in the following model element pattern:

```
(Person)[age >= 18, age <= 25]
```

As patterns are static elements, it is possible to perform various type checks at DSL compile time on them, although currently only a limited amount of type checking is done.

Pattern multiplicities are not considered to be a part of the core pattern language, but are a significant enhancement in MT over the QVT-Partners approach; they are detailed in section 3.5.

3.4. Producing target elements

When an MT rule executes, it produces one or more target elements. As this implies, MT's execution strategy is different from many 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.

3.5. 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 [29].

To solve this problem, MT adapts the concept of multiplicities found in many textual regular expression languages. Multiplicities in MT are richer than are found in graph transformation languages such as GReAT [1] which typically allow cardinality multiplicities such as `0..2` and * to be expressed. As shown in this section, MT has additional concepts such as greedy and non-greedy matching, and complete matching.

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. Among 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.
* ?	Will match against the minimum possible number of source elements.
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 pattern's appropriate argument. Whilst the `?` qualifier, in a slightly different form, is standard in most textual regular expression languages, the `!` qualifier is specific to MT.

3.5.1. 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[2]. 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]
```

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

A first glance may suggest that when the rule these patterns are a part of executes, `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. Thus the following pattern matches every dog which has a different name than its owner:

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

When the pattern matches successfully against a model element, a dictionary is created storing the variable binding names and their bound values for that pattern. This dictionary is then added to a list which records the variable bindings of all successfully matched elements. It is this list of dictionaries which is bound to the variable binding associated with the multiplicity (in the case of our example `ds`). From this list of dictionaries, variable bindings for each individual match can be accessed. To illustrate this, consider again the original multiplicities example:

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

Assuming that two model elements successfully match against this pattern, the `ds` variable would contain the following list of dictionaries:

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

Dealing with lists of dictionaries is often cumbersome. However the most common operation involves checking a condition against one of the variables bound in each dictionary of bindings in a list. The `MT` module therefore provides a generic convenience function `mult_extract(bindings, name)` (which, unlike the `transform` of section 4 is not specific to a particular transformation) which returns a list of each binding `n` in the list of dictionaries `bindings`. For example, using the previous list of dictionaries as an example, `mult_extract(..., "n")` would return `["Fred", "Barney"]`. A standard idiom is to use `mult_extract` having defined a self variable in a model element pattern. The following code uses this idiom:

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

Subsequently calling `mult_extract(ds, "d")` returns a list of all the dogs matched by the multiplicity; this can be used for further matching, or when creating target model elements.

3.6. 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`*`]]`. MT takes a slightly unusual approach to trace information. Since trace information does not play a part in the result of the transformation, the main use of tracing information in MT is for visualizing and debugging transformations. Recording every source model element that played some part in creating a given target elements often leads to huge quantities of

tracing information. Visualizing or filtering this information can be prohibitive, and so one aim of MT is to attempt to record only that trace information which is of most use to debugging.

To this end, 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. Informally, one finds 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. [29] shows how nested model element patterns in MT can be made to contribute towards tracing information if desired.

3.6.1. Augmenting or overriding the standard mechanism

While 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 for debugging purposes, or to remove certain tracing information that unnecessarily clutters the transformation visualization. MT provides a simple capability for augmenting, or overriding, the default tracing information created by the standard mechanism. Each rule can optionally specify one of the `tracing_add` or `tracing_override` clauses. These clauses contain a single Converge expression which evaluates to a tuple relating source and target model elements, and which augments or overwrites respectively the default tracing information mechanism for a given rule.

4. Example

The example I use in this paper is a variant of the standard 'class to relational model' transformation as found in [22], 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 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 that references them 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.

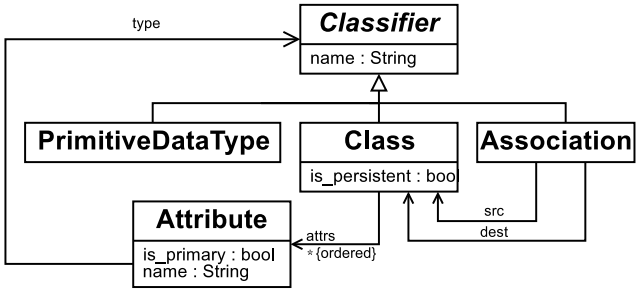

Figure 1. Extended 'Simple UML' meta-model.

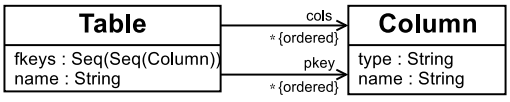

Figure 2. Extended relational database meta-model.

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

- 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:

```
import MT

$<MT.mt>:
  transformation Classes_To_Tables

  // Transform each persistent class into a table. To do this, we need to find all
  // associations for which this class is a source. We merge these associations
  // and the classes attributes together to create the columns and primary /
  // foreign keys.

  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 := []
      // We now transform the union of attributes and associations for which this
      // class is a source. Since attrs is a list and mult_extract returns a list
      // the union operator performs list concatenation.
      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)

  // Transform attributes of type String, Int etc. and which constitute part of the
  // primary key into a single column (that is both a normal column and a primary key).

  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]
```

```
// Transform attributes of type String, Int etc. and which do not constitute part of
// the primary key into a single normal column.

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]]
    []
    []

// Transform attributes of persistent types defined by users into columns. The
// transformed types primary keys become foreign keys for the table it is part of.

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)

// Transform attributes of non-persistent types defined by users into columns.

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)
```

```
// Transform associations whose destination is a persistent type into columns.
// This is analogous to Persistent_User_Type_Attribute_To_Columns.

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)

// Transform associations whose destination is a non-persistent type into columns.
// This is analogous to Non_Persistent_User_Type_Attribute_To_Columns.

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)

// Ensure that non-persistent classes etc. which are not matched by other rules do
// not lead to the creation of target model elements.

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 straightforward 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 produce 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.

Three features in this transformation need extra explanation in the context of this paper. First, the `self` variable in Converge code is analogous to `this` in Java — MT transformations are in fact translated to a Converge class, and one can thus access specific rules and so on via the `self` variable. Second, the `transform` function used throughout the transformation is also 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 also 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 final feature that requires explanation leads on from the second, 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.

5. An execution

Figure 3 shows a complete run of the example transformation on a simple input, automatically visualized as a hybrid object diagram. Source elements are shown in blue;

output elements in green; tracing links in black. An execution of this transformation with a significantly larger input model can be found in [29], which also documents MT's other visualization techniques and options.

5.1. 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. [5]) 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 [10]. 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 black 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 elements.

5.2. 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 for this example, the target model would contain many superfluous model elements. The process of removing unwanted model elements from a transformation execution is known as *model pruning* in MT.

The need for this in case of this example 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

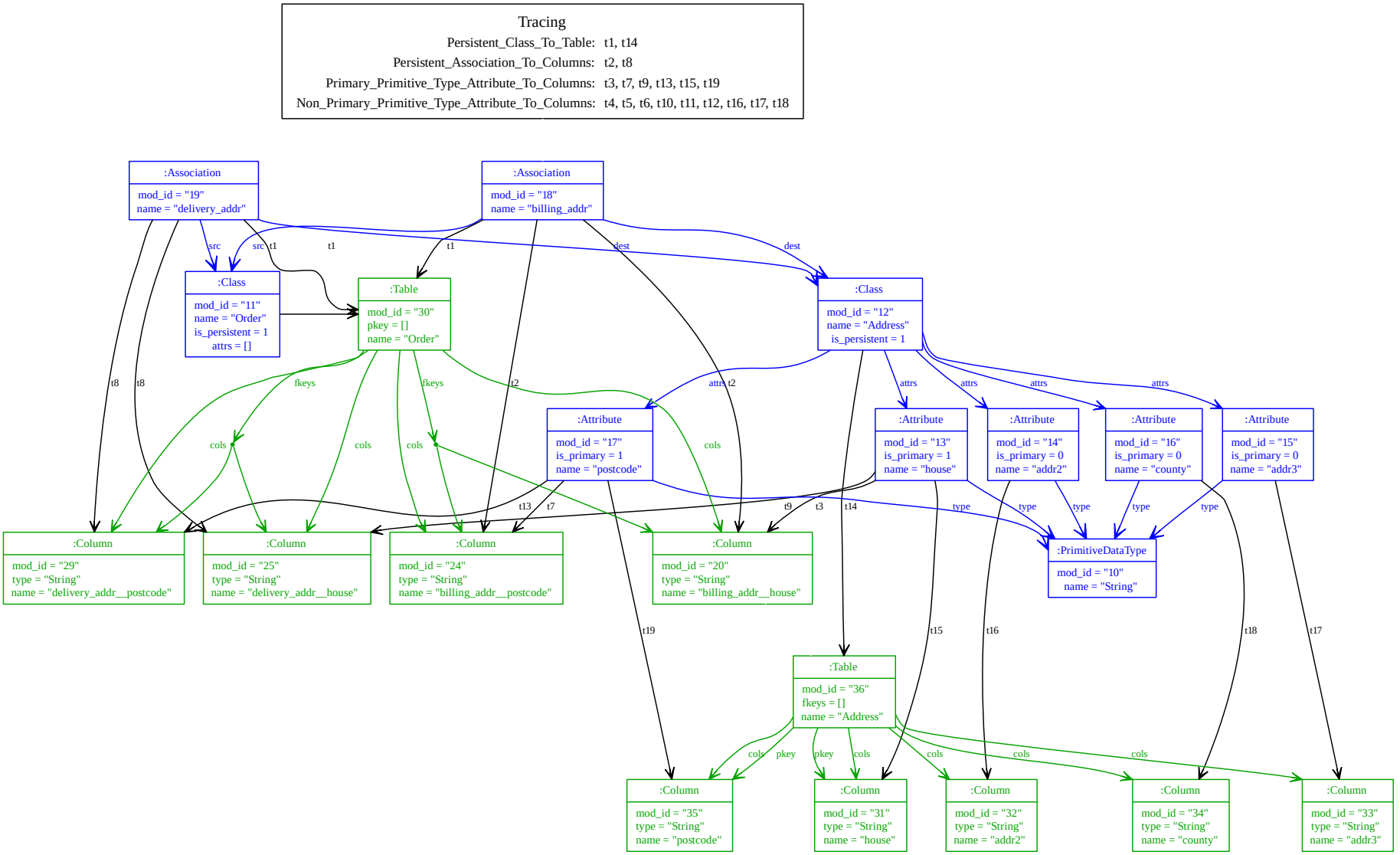

Figure 3: An example execution of the extended transformation.

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.

Figure 4 shows the output of a modified version of TM which displays model elements that would normally be pruned in dashed red. In other words, these are elements created by a rule execution (which rule can be determined, as before, from the tracing information) which are not present in the final target model. One can clearly see from this figure the need to eventually prune these elements, since they would otherwise lead to an incorrect target model.

To determine the final target model, MT uses a simple variation on the standard mark and sweep garbage collection algorithm [14]. An advantage of using this approach to model pruning is its familiarity and its quick execution time. The starting set for reachable elements in the target model are those model elements that resulted from transforming each element in the root set of source elements. From there, a simple graph walking scheme marks each model element which is reachable. Target elements not visited during this walk are then pruned. Note that this definition is carefully chosen: by examining the model elements that result from transforming each element in the root set, the eventual target model may legally consist of unconnected subgraphs.

As shall be seen in the following subsection, model pruning is also vital in the presence of combinators.

5.3. Combinators

One of the most interesting features in the QVT-Partners approach are combinators. The QVT-Partners approach defines `and`, `or`, and `not` combinators. The combinators work largely as one might expect given their names; for example, the `and` combinator takes two or more rule invocations, and succeeds only if each invocation succeeds.

Since MT rules are able to utilise the standard Converge notions of success and failure, the base combinators from the QVT-Partners approach can be encoded directly in MT using the `not`, disjunction '`|`' and conjunction '`&`' operators for `not`, `or`, and `and` respectively. The following contrived transformation rule will match against a class iff one of its attributes can be transformed by one or the other of the `R1` or `R2` rules:

```
rule X:
  srcp:
    (Class)[attributes = {<a> | 0}]

  src_when:
    self.R1(a) | self.R2(a)
```

The QVT-Partners approach defines extra semantics for the `and` combinator which automatically merges together the outputs of different rules. In the general case, I believe that such functionality is undesirable since the merging of outputs can only sensibly be determined at the fine-grained level by transformation writers themselves. However building a 'merging' combinator on top of the existing functionality is relatively simple, since it merely involves storing and then merging the result of each expression in a conjunction.

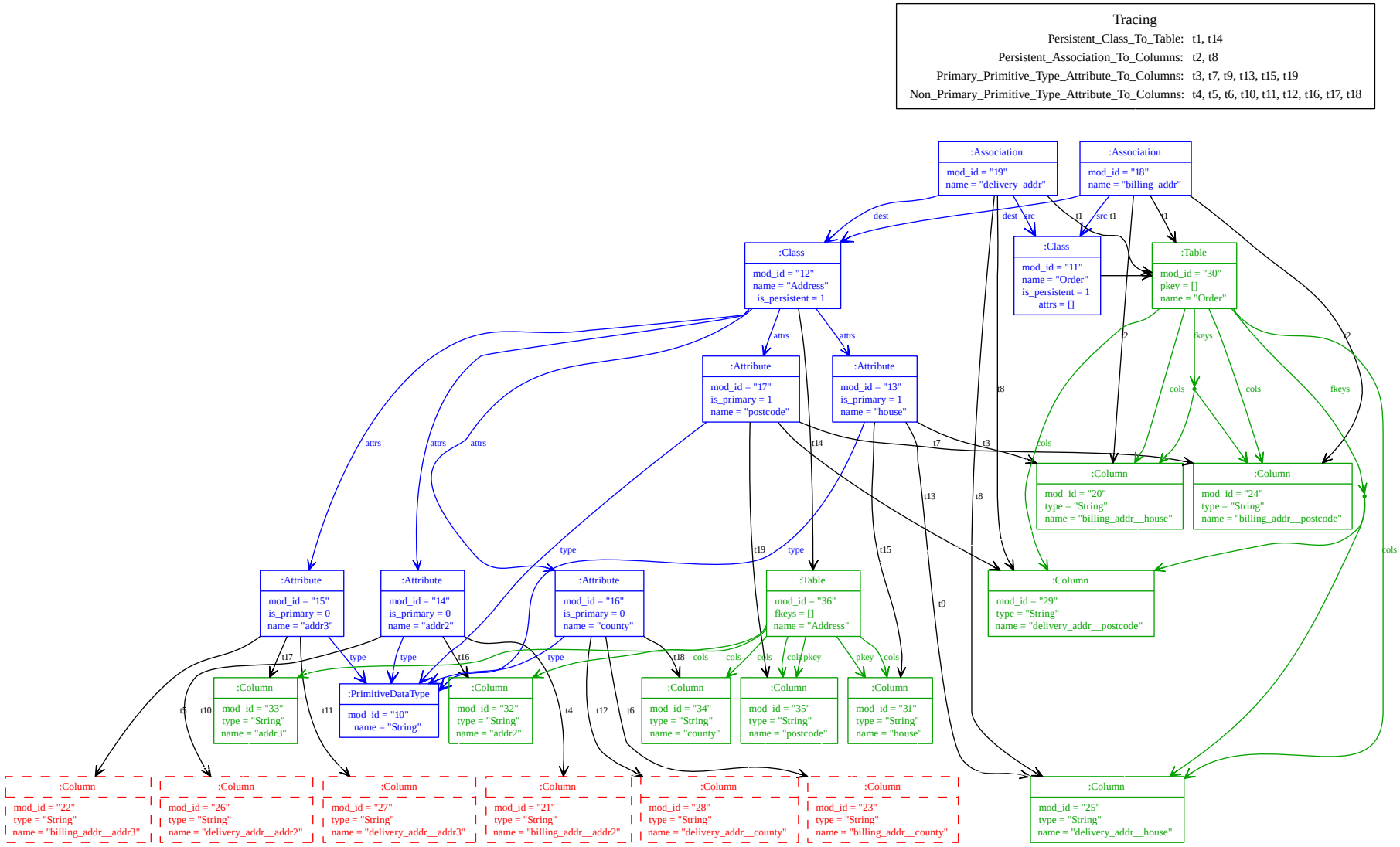

Figure 4: Non-pruned execution of the extended transformation.

Although the treatment of combinators in MT is currently simplistic, the direct encoding of these features in terms of primitive Converge features is interesting. Whereas the QVT-Partners combinators are new primitives in the language, MT is able to directly utilise Converge features.

5.3.1. Combinators and model pruning

An important, if not immediately, obvious consequence of using combinators is that model elements produced by some successful executions of rules will not be present in the final target model. This happens most frequently with the use of the `&` ('and') combinator; if its left hand side rule execution succeeds but the right hand side rule fails then elements produced by the LHS must not appear in the final target model. In fact, liberal use of the `&` combinator can easily lead to transformations which prune several times the number of the elements that appear in the final target model.

6. Related work

As with the majority of existing model transformation systems, MT is a unidirectional stateless model transformation system (in the language of [28]). MT's most obvious ancestor is the QVT-Partners approach [23]. MT takes the base QVT-Partners pattern language and enriches it with features such as pattern multiplicities, and variable slot comparisons; without such features, it is extremely difficult to express transformations of the complexity found in section 4 as discussed in [29]. Furthermore, by providing a concrete implementation – and a detailed explanation of that implementation – much of the vagueness associated with other model transformations such as the QVT-Partners approach is avoided in MT.

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. As explained in section 2.1.1, this variant language suffers from several conceptual and practical problems. The forthcoming QVT standard also includes separate 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. I also believe that MT is unique in being able to not only embed a GPL within it, but to call out naturally to that same GPL. MT users are thus not constrained by any limitations of the particular model transformation approach. Although this may initially appear to be a mere implementation detail, virtually all existing model transformation approaches present only a highly constrained execution environment to the user, requiring them to call out to a different language to augment their transformations.

Perhaps the closest model transformation approach is the commercial XMap language [6], an approach essentially based on the QVT-Partners approach. This also means that the issues noted in both this section, and in section 2.1.1 with respect to the QVT-Partners approach, apply equally to XMap.

The ATL language [3] shares many similarities with MT, including that it has a publicly available implementation. ATL does not possess patterns, or MT's advanced features such

as pattern multiplicities. ATL has an increased emphasis on the declarative nature of ATL transformations, although ATL, as MT, is a unidirectional stateless model transformation language. [16] shows the ATL version of the example of section 4. Mainly due to its lack of patterns and so on, the ATL transformation is approximately a little over a third larger than the MT equivalent; it is also more complex than the MT equivalent due to it use of ATL's three different types of rule (standard, lazy, and unique lazy), where MT only has one type. However ATL does have some useful additional features such as the notion of rule inheritance which can be used to provide an analogue to method overloading in an OO language; MT has no such equivalent.

Graph transformation approaches, such as that presented in [18] take a very different approach to MT (and, indeed, ATL). Graph transformations are more declarative in nature than MT, and allow more theoretical reasoning about their transformations than is practical with MT. While many of the graph transformation approaches mentioned in this paper are fairly mature, MT has a richer pattern language – particularly with its treatment of pattern multiplicities – in comparison to languages such as GReAT [1] and PROGRES [24], and a very different approach to embedding a GPL within the language. VIATRA2 uses abstract state machines which to express complex control flow [2]; while powerful, it is rather verbose compared to MT's use of the embedded Converge language. One advantage of some graph transformation approaches is that they have accompanying implementations which are mature and stable (e.g. [18,20]).

The RubyTL language [7] is similar in spirit to MT, in that it is a model transformation language expressed as a DSL within a GPL. Although Ruby has a more flexible syntax than most languages, it does not permit arbitrary syntaxes to be embedded within it as does Converge. Most notably this means that RubyTL does not possess syntactically rich patterns, relying on 'filters' (effectively `when`-style clauses) written in normal imperative code.

A number of other approaches have also been published using a similar example to that of section 4. Diagrammatic approaches such as MOLA are difficult to use on such 'fiddly' examples, and can often lead to transformations which are more difficult to comprehend than textual approaches such as ATL and MT [17]. Other approaches such as BOC appear to lack an implementation, making it hard to be sure whether pseudo-code examples are executable or not [19]; and in the case of BOC, the pseudo-code for the transformation of section 4 is more than twice as long as the MT equivalent, raising questions about the practicality of the approach.

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 [9], require the user to manually specify the tracing information to be created. By 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.

The unnamed language presented in [28] is effectively a persistent model transformation language in the spirit of the original QVT-Partners proposal. Although it can express persistent transformations, it is therefore a much cruder and less expressive language language than MT.

It is perhaps telling that although MT contains several enhancements compared to existing approaches, it also shares many of the limitations of existing approaches, such as a lack of rule structuring mechanisms. Section 7 outlines the work that may resolve some of these limitations.

7. Future work

Although I believe that MT is currently one of the more powerful 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 rules in an MT transformation exist in a single namespace; there is no notion of 'transformation modules'. The issue of scalability is perhaps the most crucial in maturing model transformations as an area.

Currently MT performs virtually no optimisations. That is, it is possible for the user to express inefficient transformations that MT could relatively easily transform into equivalent, efficient transformations. For example it should be possible to analyse patterns in a `srcp` clause and determine an equivalent reordering that will allow the transformation to execute quicker.

A little explored area of MT at current is combinators. I believe that a fruitful area of research will be to build upon the work of Section and to build and experiment with increasingly powerful combinators.

In terms of 'nitty gritty' details, there are several aspects of MT that could usefully be improved. For example, one irritation encountered in this paper relates to the `for` suffix of expressions in a rules `tgtp` clause. Currently rules can generally only produce as many top-level elements as they have expressions in the `tgtp` clause. This can occasionally lead to cumbersome or dangerous work arounds being employed. It would be useful to have a variant `for` suffix which would 'fold in' the elements produced by its expression as if they had been produced by top-level `tgtp`. As befits a new, small language similar examples can easily be found elsewhere in MT.

MT has been used as the basis for a change propagating model transformation approach, which raises a number of new challenges; this will be documented in a follow up paper.

8. 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 pruning extraneous elements. I then made use of these novel features in a non-trivial example.

Given the relative immaturity of the model transformations area, it would be naïve to assert that any particular approach is the definitive answer. Rather I hope that the description of MT and its novel features provides a useful basis for authors of subsequent approaches, as the model transformation community researches different ways to tackle

this vital problem. Furthermore I believe that MT shows that the Converge language is a practical way of implementing such languages quickly and robustly.

An extended version of this paper can be found in the technical report [29]. MT can be found as part of the Converge programming language, freely available under a MIT / BSD-style licence from `http://convergepl.org/`.

I would like to thank the anonymous referees whose comments on this paper were extremely helpful.

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

REFERENCES

1. A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. *The design of a language for model transformations*, volume 5. September 2006.
2. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pages 1280–1287, Dijon, France, April 2006. ACM Press.
3. J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.
4. J. Bézivin and S. Gérard. A preliminary identification of MDA components. In *Generative Techniques in the context of Model Driven Architecture*, Nov 2002.
5. H.-D. Böcker and J. Herczeg. What tracers are made of. In *Proc. ECOOP '90*, pages 89–99, 1990.
6. 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.
7. J. S. Cuadrado, J. G. Molina, and M. M. Tortosa. RubyTL: A practical, extensible transformation language. In *ECMDA-FA*, pages 158–172, 2006.
8. K. Czarnecki and S. Helsen. Classification of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
9. DSTC, IBM, and CBOP. MOF query / views / transformations first revised submission, August 2003. OMG document `ad/2003-08-03`.
10. A. Egyed. A scenario-driven approach to traceability. In *Proc. 23rd International Conference on Software Engineering*, pages 123 – 132, 2001.
11. 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`.
12. 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.
13. R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.
14. R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Mem-*

ory Management. Wiley, 1999.
15. S. P. Jones. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press, April 2003.
16. F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138, October 2005.
17. A. Kalnins, E. Celms, and A. Sostaks. Model transformation approach based on mola. In *Model Transformations in Practice workshop*, October 2005.
18. A. Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice workshop*, October 2005.
19. M. Murzek, G. Kappel, and G. Kramler. Using the boc model transformer. In *Model Transformations in Practice workshop*, October 2005.
20. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE)*, pages 742–745. ACM Press, 2000.
21. Object Management Group. *Meta Object Facility (MOF) Specification*, 2005. `formal/05-05-05`.
22. QVT-Partners. First revised submission to QVT RFP, August 2003. OMG document `ad/03-08-08`.
23. QVT-Partners initial submission to QVT RFP, 2003. OMG document `ad/03-03-27`.
24. A. Schürr, A. J. Winter, and A. Zündorf. *Handbook of Graph Grammars and Graph Transformation*, volume 2, chapter PROGRES: Language and Environment, pages 487–550. 1999.
25. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.
26. L. Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proceedings Dynamic Languages Symposium*, pages 49–64, October 2005.
27. L. Tratt. The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London, February 2005.
28. L. Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.
29. L. Tratt. The MT model transformation language. Technical Report TR-05-02, Department of Computer Science, King's College London, May 2005.
30. L. Tratt. The MT model transformation language. In *Proc. ACM Symposium on Applied Computing*, pages 1296–1303, April 2006.
31. G. van Rossum. Python 2.3 reference manual, 2003. `http://www.python.org/doc/2.3/ref/ref.html` Accessed Aug 31 2005.