The MT model transformation language

Technical report TR-05-02, Department of Computer Science, King’s College London

Laurence Tratt
laurie@tratt.net

May 11, 2005

Contents

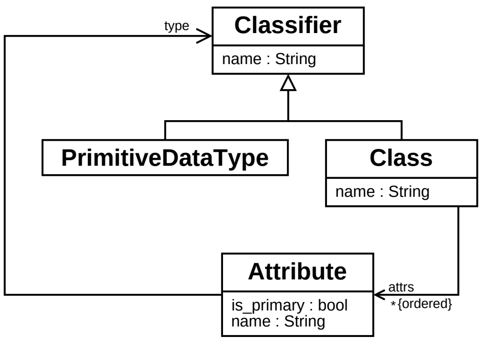


Figure 1: ‘Simple UML’ meta-model.

1. Introduction

This paper presents a unidirectional stateless model transformation language MT, implemented as a DSL within Converge. MT shares several aspects in common with model transformation languages such as the QVT-Partners approach [QVT03b]. That is it is a rule-based system, utilising patterns. However there are a number of advances over, and significant differences from, previous approaches. Some of these are a side-effect of implementing MT as a DSL within Converge; some are the result of experimentation with a concrete, but malleable, implementation. For example, MT allows normal Converge imperative expressions to be embedded within it. In a wider context, MT is used as the basis for a change propagating transformation language which will be presented in a later paper.

This paper comes in three main parts. Firstly the running example is introduced, followed by an introduction to the QVT-Partners model transformation approach. The QVT-Partners approach is then used as the basis for the MT language, which is introduced partly through example. Finally the implementation of the MT language as a DSL in Converge is discussed in more detail.

2. Running example

This paper makes use of a simple running example of a transformation from a UML like modelling language to a model of relational databases. The chief reason for using this example is the ability to compare the result with its implementation in other model transformation approaches (e.g. [DIC03, OQV03, QVT03a]. The example also has the virtue that it can be easily considered in ‘simple’ and ‘advanced’ variants. Whilst the advanced variant does exercise a reasonable number of features of any given model transformation approach, it is still of an appropriate size for this paper.

The original example is defined in [QVT03a]. The simple variant is as follows. A meta-model for a simple UML modelling language is given in figure 1. A corresponding meta-model for simplified relational databases is given in figure 2. In essence, the transformation takes in a `Class` and transforms it to a `Table` of the same name. `Attribute`’s whose type is a `PrimitiveDataType` (e.g. string, integer) are transformed to a column of the same name and the same primitive data type. `Attribute`’s whose type is a `Class` (i.e. the type refers to another user class) are transformed to a number of columns: the transformation recursively ‘drills down’ through a class’s non-primitively typed attributes until it reaches attributes with primitive datatypes. At each point in the recursion the name of the current class along with a ‘_’ character is appended to the column name. The net result of this is that non-primitive data types are flattened.

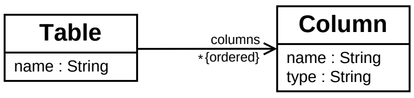


Figure 2: Simplified relational database meta-model.

Consider the source model of figure 3. Assuming that the `Dog` class is used as the input class, an implementation of this transformation should produce the relational database model as in figure 4.

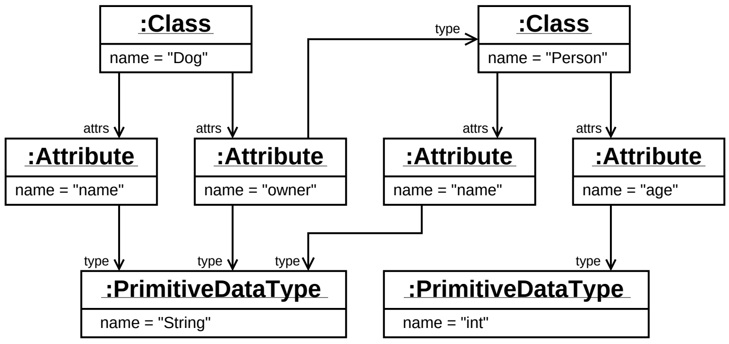


Figure 3: An example of a source simple UML model.

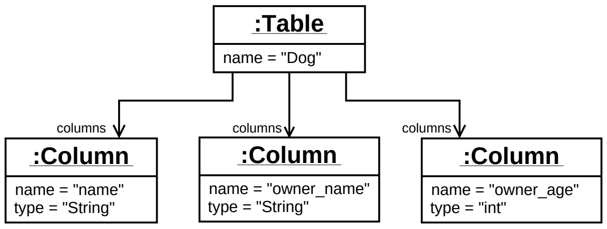


Figure 4: An example of a target relational database model

This is the core of the example that will be used throughout this paper. As the paper progresses, I will progressively add complexity to the example.

3. The QVT-Partners model transformations 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 (known as *relations* and *mappings* respectively[1]), for the purposes of this paper, transformation specifications are largely irrelevant and are consequently ignored. The QVT-Partners approach also defines a diagrammatic syntax for transformations which is similarly ignored.

3.1. Overview

A transformation in the QVT-Partners approach consists of a number of mappings. A mapping consists of one or more source *domains* (analogous to a function parameter) and a target imperative body. Each domain

[1]The author of this paper takes full responsibility for his decision to use these names — given their multiple overloaded meanings in the wider fi eld, in retrospect they were not the best possible choices.

consists of one or more *patterns* to match against. Patterns are written in a language designed to make expressing constraints over models succinct; they are analogous to textual regular expressions as found in e.g. Perl [WCO00]. Imperative bodies consist of a single expression in an extended OCL variant that is capable of side effects. Using the meta-models presented in the previous section, a simple mapping for transforming a class to a table would look as follows:

```
mapping Class_To_Table {
  domain {
    (Class)[name = n, attrs = A]
  }

  body {
    let
      columns = A->collect(attr columns = Set{} |
        columns + Attr_To_Column(attr))
    in
      (Table)[name = n, columns = columns]
    end
  }
}
```

The intuitive meaning of this is hopefully fairly straightforward. The `Class_To_Table` rule will match against a `Class` model element, with the pattern binding whatever name the class has to the variable `n` and whatever attributes it has to the variable `A`. The imperative body then creates a corresponding `Table` whose name matches the source class. It should be noted that although the `Table` expression in the body appears to be a pattern, this is something of a syntactic illusion: the pattern-esque syntax is simply syntactic sugar for object creation and slot updating. `Attr_To_Column` refers to another mapping, which is used to transform each attribute in the source class into one or more database columns which are then placed within the target table.

3.2. Pattern language

In the context of this paper, the most important novel aspect of the QVT-Partners approach is its pattern language. Its aim is to provide a concise textual notation for expressing constraints over models, thereby reducing the time needed to write and to comprehend a transformation. In this subsection, I provide a brief background of patterns before informally explaining the QVT-Partners pattern language.

Many computer users are familiar with textual pattern languages e.g. via operating system commands such as `ls *.txt`. One can obtain a crude gauge of the popularity of textual regular expressions by the fact that suitable libraries are found as standard in many modern programming languages such as Perl, Python and Ruby. However whilst pattern languages are commonly thought of as being suitable only for matching against text, they can be used to match against other, much richer, datatypes. For example program transformation patterns match against complex AST's [Big98]. Intuitively, designing a pattern language involves a compromise between providing a concise notation for capturing common constraints, and providing a completely general mechanism — the more cases a pattern language can express, the less concise it is likely to be. Many pattern languages are thus tailored for the common case as opposed to the general case. Textual regular expressions, for example, are typically defined as finite-state automata which can not express a seemingly simple constraint such as ensuring that a string contains balanced open and close brackets[2]. It is therefore desirable that when a pattern languages' expressive limit is reached, a suitable escape mechanism into a more powerful, if verbose, system is available.

[2]Note that some implementations of regular expressions are no longer 'regular' in the formal sense of that word. For example, modern Perl contains an experimental feature which can express the balanced brackets constraint.

The QVT-Partners approach provides a specific pattern language for expressing constraints over models. By providing many of the benefits known to textual regular expressions users, several of the problems concerned with identifying elements in a model are addressed. The QVT-Partners approach is important in this respect because most current model transformation approaches do not provide pattern languages. Although it could be argued that graph transformation approaches utilise pattern languages, I believe that these lack any significant expressive power, particularly when compared to textual regular expressions; for the purposes of this paper, I therefore do not consider graph transformation approaches to have pattern languages.

The QVT-Partners approach provides a small pattern language for expressing constraints over models. A slightly simplified version of the grammar for the pattern language is as follows:

```
⟨pattern⟩ ::= '_'
  |  ⟨set_pattern⟩
  |  ⟨seq_pattern⟩
  |  ⟨obj_pattern⟩
  |  ⟨expression⟩

⟨set_pattern⟩ ::= '{' [⟨pattern⟩* ('|' ⟨pattern⟩)*] '}'

⟨set_pattern⟩ ::= '[' [⟨pattern⟩* ('|' ⟨pattern⟩)*] ']'

⟨obj_pattern⟩ ::= '(' ⟨var⟩ [ ',' ⟨var⟩ ] ')' '[' [⟨field_pattern⟩ (',' ⟨field_pattern⟩)*] ']'

⟨field_pattern⟩ ::= ⟨var⟩ '=' ⟨pattern⟩

⟨var⟩ ::= 'ID'
  |  '_'
```

The reference to the rule *expression* is a reference to a rule which contains the extended OCL variants' grammar.

The QVT-Partners approach identifies three main types of patterns: set, sequence, and object patterns. Although not explicitly noted as such, variables in patterns are essentially patterns themselves. To ensure consistency with the rest of this paper, I refer to object patterns as *model element patterns*. 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 definition of object equality in the QVT-Partners approach is inherited from the MOF [OMG00]). The special variable '_' matches against anything and immediately discards the result; multiple instances of '_' do not need to match against equivalent objects.

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 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. This pattern is approximately equivalent to the following Converge-esque pseudo-code function which returns a dictionary of bindings if the source element is matched successfully, failing otherwise:

```
func match(element):
  if not element.conforms_to(Dog):
    return fail
  d := element
  n := element.name
  if not element.owner.conforms_to(Person):
    return fail
  if element.owner.name != "Fred":
    return fail

  return Dict{"d" : d, "n" : n}
```

Although it may seem more logical to have used OCL to express this, it should be noted that expressing the creation and update of bindings in OCL would require complex encodings. Partly due to this difficulty, the QVT-Partners approach defines a new calculus in order to have a suitable semantic domain. The calculus is given an operational semantics, and directly implements several pattern matching primitives; it can be seen to be similar to the imperative object calculus of Abadi and Cardelli [AC96] extended with pattern matching. Using Converge pseudo-code as the target translation of the example pattern avoids the need to define and explain the calculus.

As this example translation clearly shows, the model element pattern is not only considerably terser than its equivalent pseudo-code, but is arguably easier to comprehend. Since the pseudo-code has to explicitly embed certain aspects of the model transformation (e.g. the `return fail` statements) the important aspects of the pattern are obscured. This is simply a recasting of the problem of expressing model transformations in GPL's. Even though the pattern language is simple, it neatly solves many such problems.

3.3. Complete example

The running example expressed in the QVT-Partners approach is as follows:

```
mapping Class_To_Table {
  domain {
    (Class)[name = n, attrs = A]
  }

  body {
    let
      columns = A->collect(attr columns = Set{} |
        columns + or(Primitive_Type_Attr_To_Column("", attr),
          User_Type_Attr_To_Column("", attr))
    in
      (Table)[name = n, columns = columns]
    end
  }
}

mapping User_Type_Attr_To_Column {
  domain {
    (String, prefix)[]
  }

  domain {
    (Attribute)[name = n, type = (Class)[name = ct, attributes = A]]
  }

  body {
    let
      new_prefix =
        if prefix == "" then
          n
        else
          prefix + "_" + n
        end
    in
      for A->collect(attr attrs = Set{} |
        attrs + or(Primitive_Type_Attr_To_Column(new_prefix, attr),
          User_Type_Attr_To_Column(new_prefix, attr)))
    end
  }
}

mapping Primitive_Type_Attr_To_Column {
  domain {
    (String, prefix)[]
  }

  domain {
    (Attribute)[name = n, type = (PrimitiveDataType)[name = pt]]
  }

  body {
    Set{(Column)[name = if prefix == ""
        name = n
      else
        name = prefix + "_" + n
      end,
      type = pt]}
  }
}
```

One feature in particular that requires explanation is the `or` function used in the `Class_To_Table` and `User_Type_Attr_To_Column` mappings. `or` is not a normal function call, but is a built-in *combinator* which lazily executes the mappings passed as arguments to it in order until one succeeds and produces a value. Note that unlike most rule-based systems, the QVT-Partners does not provide a function which takes an element and attempts to find a rule which will transform it. Although the `or` combinator can provide the same functionality, its repeated use becomes tiresome due to the continuous hard-coding of mapping names required.

The overall structure of this transformation is fairly simple. The `Class_To_Table` is the top-level mapping which takes in a a class and iterates through its attributes, invoking other mappings to produce columns. Attributes are transformed in one of two ways. Both the `User_Type_Attr_To_Column` and `Primitive_Type_Attr_To_Column` mappings take two arguments: a string and an attribute. The string represents the current column name prefix being built up as the transformation drills into user data types. Attributes which have a primitive data type are transformed by the `Primitive_Type_Attr_To_Column` mapping into a single column. Attributes which have a user data type are transformed by the `User_Type_Attr_To_` into one or more columns; the `User_Type_Attr_To_Column` mapping is the recursive mapping which drills into user data types.

Although the example from section 2 has been successfully expressed in the QVT-Partners approach, the result is perhaps more verbose than one may have expected. Indeed, somewhat surprisingly, a simple GPL equivalent of this example is smaller. One might thus reasonably expect that expressing such a transformation in the QVT-Partners approach has other benefits. Since mappings only allow the expression of unidirectional stateless transformations, the only potential gain over a GPL approach is the possibility of automatically created tracing information. Unfortunately the QVT-Partners approach does not explain explain how rules can create such tracing information in practise. Since the GPL equivalent is likely to be more readily understood by a far wider range of people, the overall benefits of this approach are not clear cut. In the following subsection I outline three issues which are indicative of the problems of the QVT-Partners approach.

3.4. Issues with the approach

As the verbose example in section 3.3 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, three areas which are indicative of where the QVT-Partners falls short of its intended goals. These points are instructive in understanding several of the design decisions made in the MT language of section 4.

3.4.1. 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[3]. Conversely when it comes to acting as a normal GPL, the resulting language is decidedly 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 be achieved clumsily via the `let` expression.

3.4.2. Underpowered patterns

The pattern language defined in the QVT-Partners approach is novel in the context of model transformations, and potentially very useful. However as the relatively simple definition in section 3.2 may suggest the pattern language is lacking in significant expressive power.

The pattern language itself is limited in two main ways:

1. Within model element patterns it is only possible to check for the equality of slots. For example, 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.

   In order to sidestep this problem, users must add additional OCL in a `when` clause.

2. Model element 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.

[3]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.

3.4.3. 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. Consider the simple example of section 3.2 `(Dog, d)[...]`. `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 means that it is impossible to express that a model element pattern should match against a particular element. For example, in a meta-circular system where a class `M` is an instance of the `Singleton` meta-class, the model element pattern `(Singleton, M)[...]` will create a local variable `M` rather than ensuring that it matches against the element pointed to by `M`.

The QVT-Partners approach allows a `when` clause to be scoped over all domains establishing a constraint across domains since it does not create any new variable bindings. However it is not possible to introduce a similar clause (often called `where` in similar approaches) scoped over all domains which introduces new variable binding without introducing ambiguities. Consider the following example: should the `x` in the pattern bind the value of `slot` to the variable, or should it ensure that `slot` contains the value introduced in the `where` clause?

```
mapping X {
  domain {
    (E)[slot = x]
  }

  where {
    x := 5
  }
}
```

In the QVT-Partners approach, scoping ambiguities are avoided by disallowing several potentially useful features that may introduce new variable bindings. However the end result is that whilst expressions such as in `(Dog, d)[...]` are statically resolvable, they are confusing for users. The overall effect of the scoping rules are to severely limit the possibilities for extending or embedding the language.

3.5. Summary

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 GPL's.

In the next section of this paper, I define a new model transformation language MT which takes elements of the QVT-Partners approach, adding extra features and addressing some of the approaches flaws.

4. The MT Language

The MT language is a new unidirectional stateless model transformation language, implemented as a DSL within Converge. MT transforms instances of the typed modelling language TM [Tra05] into new instances of TM. In essence, 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 a concrete implementation from shortly after its original design was sketched out. This has proved to be significant, since practical experience with the approach has been rapidly fed back into the implementation. Rapid development has been facilitated by the flexible environment provided by Converge. The ability to experiment with the implementation has ultimately led MT to contain a number of insights and distinct differences from other approaches. Such insights range from a more sophisticated pattern language to suitable ways to visualize model transformations. In the wider context of this paper, MT is also important as the basis of a change propagating language, to be described in a later publication.

In this section I first highlight the main features of MT, then present an MT version of the running example, before showing how MT transformations are run in practise. This section is intended to present the basic features of MT, before more advanced features are described in section 6.

4.1. 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. Rules and functions in MT are essentially synonymous as in approaches such as TXL [Cor04]. 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, nothing is 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 no rule matches a given element, 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 `srcp` and `srcp_when` clauses are collectively said to form the `source model clauses`; similarly the `tgtp` and `tgtp_when` clauses are collectively said to form the `target model clauses`.

Transformations are translated by MT into a Converge class with the name of the transformation; rules are translated to functions of the same name within the class. In order to run a transformation, the transformation class is instantiated; each class can be instantiated multiple times. Transformation classes have additional functions for e.g. extracting tracing information (see section 4.6).

Transformation rules contain normal Converge code in expressions; such expressions can reference variables outside of the model transformation DSL fragment. This is an important aspect of MT since it allows users to use normal Converge functions arbitrarily, and without penalty. In other words, when the model transformation language itself is inadequate in a particular respect, a normal reusable Converge function can be defined outside of the model transformation, but which can be called from within any model transformation.

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, encoded as a Converge list, of the form `[[`*`source elements`*`], [`*`target elements`*`]]`. 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. By default, only elements matched by non-nested model element patterns are recorded in the tracing information. Section 5.2 details the default tracing creation mechanism, and explains how it can be augmented or overridden.

A simple example of a transformation and a rule is as follows:

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

  rule Class_To_Table:
    srcp:
      (Class, <c>)[name == <n>, attrs == <A>]

    tgtp:
      (Table)[name := n, cols := columns]

    tgt_where:
      columns := []
      for attr := A.iterate():
        columns.extend(self.transform([""], [attr]).flatten())
```

This rule is the MT analogue of the rule in section 3.1. Note how normal Converge code is interspersed amongst the MT DSL (see section 7.13 for implementation information). Since transformations are translated to Converge classes, to access functions 'internal' to the transformation, one must use the `self.` prefix. The `transform` function, for example, takes source elements and tries each rule in the transformation in succession until it finds one which successfully matches the elements and produces values. If the `transform` function does not find a suitable rule then, by default, an exception is raised. The `transform` function is also used internally by the transformation as the mechanism used to transform the root set of source model elements. Section 6.3 shows an example of a rule that can guarantee that the `transform` rule can be made to succeed on any given input.

In the following subsections I explain in more detail how rules match and produce elements, including a detailed examination of the pattern language and pattern multiplicities.

4.2. Matching source elements with patterns

Each pattern in the `srcp` clause of a rule corresponds to a domain in the QVT-Partners approach. Arguments must be passed as lists rather than sets; whilst TM elements can be placed into Converge sets, users may wish to transform non-hashable elements such as lists. Each list contains the top-level model elements which each pattern can match against. Elements can exist, directly or indirectly – that is as top-level elements, or by being reachable via the graph that constitutes a model – in one or more arguments. In order to avoid the problems noted in section 3.4.3, variable bindings are surrounded by angled brackets '<' and '>' to distinguish them from normal Converge variable references.

The matching algorithm used by MT is intentionally simple. Each pattern 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 called), which attempt to generate another match given the variable bindings and arguments available to them. The generation of an alternative match causes new variable bindings to produce, which allows the rule to attempt another match of later patterns. The `src_when` clause, if it exists, is tried 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. The implementation details of such behaviour are largely hidden from the user by the use of patterns.

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.

Each pattern is translated to a Converge generator, which provides a natural mechanism for lazily generating all possible matches. Translated patterns are co-joined to make use of Converge's backtracking abilities. Note that the `src_when` clause, if it exists, must be a single Converge expression which either succeeds or fails given variable bindings generated by patterns in the `srcp` clause.

4.3. Pattern language

MT's pattern language is essentially 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. Patterns written in the latter language can be directly translated into MT with only minor syntactic changes.

There are two significant differences between the two pattern languages. Firstly – as noted in the section 4.2 – variable bindings in MT must be surrounded by angled brackets to ensure harmony between MT and Converge's scoping rules. Secondly, model element patterns in MT can contain comparisons other than equality between slots; henceforth these are known as *slot comparisons*. Any of the standard Converge comparison operators can be used in slot comparisons. A model element pattern in MT is said to consist of zero or more slot comparisons.

As a trivial example of slot comparisons, one can take the model element pattern example from section 3.2 (making the necessary minor syntactic modifications), and change it to find dogs whose owner is not Fred:

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

This same example would necessitate an OCL constraint in a `when` clause in the QVT-Partners approach.

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]
```

There is one other additional feature in the MT pattern language. Since model elements are Converge objects, slot comparison is not entirely synonymous with attribute comparison, since slots may contain functions. MT's model element patterns therefore provides support for functions as shown in the following example:

```
(Person)[calc_wage() > 18000]
```

Functions in slot comparisons can be passed an arbitrary number of arguments passed to them; all arguments are normal Converge expressions.

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 6.2.

4.4. Producing target elements

When an MT rule executes it produces one or more target elements. An exception is raised if a rule executes but fails to produce any elements. The number of elements produced is determined by the number of expressions in the `tgtp` clause. If the `tgtp` clause has a single expression, then the rule produces a single element; if it contains more than one expression, then the rule produces a list whose length is the same as 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 the Converge grammar rule `expr` (see section 7.14 for implementation details). Model element expressions differ from model element patterns both conceptually and syntactically. Conceptually a model element expression is an imperative, creational action. There is therefore no concept of a 'self' variable in a model expression. Furthermore, to reinforce the notion that model expressions are imperative actions, slot assignments 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 its normal Converge counterpart, taking a single expression and continuously pumping it for values until it fails. Variables defined in the `for` suffix are scoped only over the single expression in the `tgtp` clause that it suffixes. Assuming `COLS` is a list, a typical usage of this feature is as follows:

```
(Column)[name := col.name] for col := COLS.iterate()
```

Note that if the above example was the only expression in a `tgtp`, the result of the rule would be a list of length `COLS.len()`. However, if the expression was the first of two in a `tgtp`, the rule would produce a list of length two, with the first element being a list of length `COLS.len()`. Section 9 suggests a possible extension to MT which would allow a rule to produce a number of elements not solely determined by the number of expressions in its `tgtp`.

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.

4.5. Example

The following is a complete Converge module which implements the running example:

```
import Sys
import Relational, Simple_UML
import MT.MT

func concat_name(prefix, name):
  if prefix == "":
    return name
  else:
    return prefix + "_" + name

$<MT.mt>:
  transformation Classes_To_Tables

  rule Class_To_Table:
    srcp:
      (Class, <c>)[name == <n>, attrs == <A>]

    tgtp:
      (Table)[name := n, cols := columns]

    tgt_where:
      columns := []
      for attr := A.iterate():
        columns.extend(self.transform([""], [attr]).flatten())

  rule User_Type_Attr_To_Column:
    srcp:
      (String, <prefix>)[]
      (Attribute)[name == <n>, type == (Class)[name == <cn>, attrs == <CA>]]

    tgtp:
      self.transform([concat_name(prefix, n)], [ca]) for ca := CA.iterate()

  rule Primitive_Type_Attr_To_Column:
    srcp:
      (String, <prefix>)[]
      (Attribute)[name == <n>, type == (PrimitiveDataType)[name == <pn>]]

    tgtp:
      [(Column)[name := concat_name(prefix, n), type := pn]]
```

The overall structure of this transformation is deliberately similar to the version in the QVT-Partners approach of section 3.3. One important difference is that the repetitive code which builds up the column prefix is factored out into a normal top-level function `concat_name` — this contributes towards making the transformation approximately 10% smaller than the QVT-Partners approach equivalent.

A slight difference between the MT transformation and the QVT-Partners approach equivalent is that the `User_Type_Attr_To_Column` rule produces a list which contains list of columns. The outer list will be of length `CA.len()`, with each entry in the list being of arbitrary length. Consequently the `flatten` function call in line 24 is necessary to remove the nesting that will be present if the `User_Type_Attr_To_Column` rule is called.

4.6. Running a transformation

Details of how to run a model transformation, including details such as the format of its inputs and outputs and so on, are surprisingly absent from descriptions of the majority of model transformation approaches. Since this is one of the most important practical aspects of model transformations, I believe it is important to be explicit about how transformations are run. In this subsection I detail the process of running a MT transformation.

Running a transformation in MT involves instantiating a transformation class and passing it model elements. The transformation then executes, attempting to find a rule to transform each element in the root set of source elements. If the transformation is successful in transforming the root set of elements, a transformation object will be returned. The transformation object can then be queried to find the target model elements produced and the corresponding tracing information. The format of MT's inputs and outputs is simple. Source elements must be instances of elements defined in a `TM.model_class` block (note that built-in Converge types such as strings and ints are defined to be valid TM model elements). Similarly target elements will be TM model elements.

The following example creates a simple input model and then executes the `Classes_To_Tables` transformation:

```
dog := Simple_UML.Class("Dog")
person := Simple_UML.Class("Person")

dog.attrs.append(Simple_UML.Attribute("name", Simple_UML.String))
dog.attrs.append(Simple_UML.Attribute("owner", person))

person.attrs.append(Simple_UML.Attribute("name", Simple_UML.String))
person.attrs.append(Simple_UML.Attribute("age", Simple_UML.Integer))

transformation := Classes_To_Tables(dog)
```

The target elements produced by the transformation can be accessed via the `get_target` function. Since both the source and target elements are TM model elements, one can apply the standard TM visualization to our example. The source model is shown in figure 5, with the target model in figure 6. Note that the colours given to the source and target models will be used in the remainder of this paper: source elements are shown in blue, target elements in green.

If an element passed to the `transform` function can not be transformed by any of the available rules, an exception is raised showing the offending element(s) and the transformation is aborted. Users may catch such an exception if desired, however one may reasonably ask why the transformation does not attempt to recover gracefully in such instances. Unfortunately this seems to be unrealistic in the general case for the following reason. Since the `transform` function is called with the expectation that it will return a result, when it fails to find a suitable rule to transform a given element it is unable to fulfil the callers expectation that an element will be returned. In order to maintain this expectation, `transform` could conceivably return a 'dummy' target element as a placeholder. However such a dummy element would be unlikely to satisfy the constraints on the target meta-model, and would thus generally cause an exception to be raised.

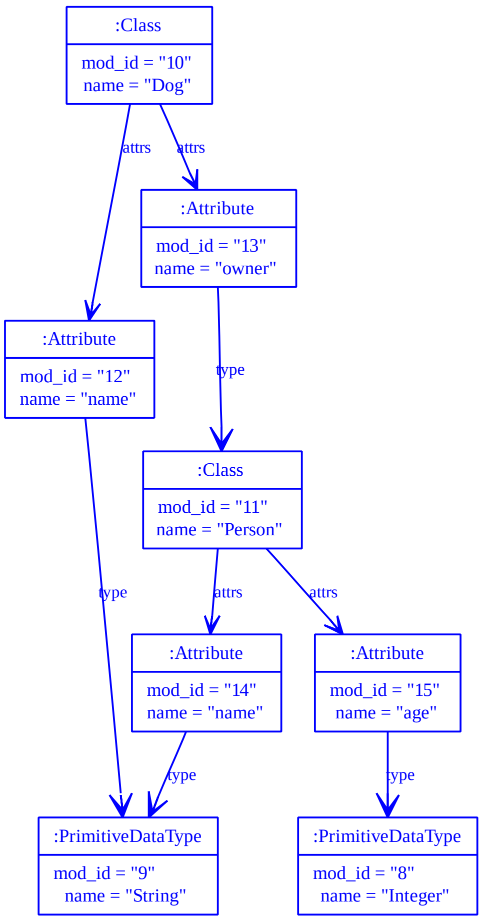


Figure 5: Source model.

In the, probably small, number of situations where the dummy element did not cause an error, it is then less than clear that the resulting target model will be of significant use to the user. Section 6.3 shows an example of a 'default' rule which guarantees that the `transform` function can not fail.

5. Tracing information

In this section I describe how MT deals with tracing information. First I show how tracing information is visualized, then describe the standard mechanism for creating it, before showing how the user can augment or override the default tracing information created.

5.1. Visualizing tracing information

Section 4.6 showed how a MT transformation can be run, and used the default visualization capabilities of TM to visualize the source and target models of a transformation. However, MT transformation instances also store tracing information (see section 4.1) relating source and target elements. Visualizing tracing information is an interesting challenge, and one that has hitherto received scant attention in the setting of model transformations. 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 order to visualize tracing information, one needs to understand how this information is stored. Trans-

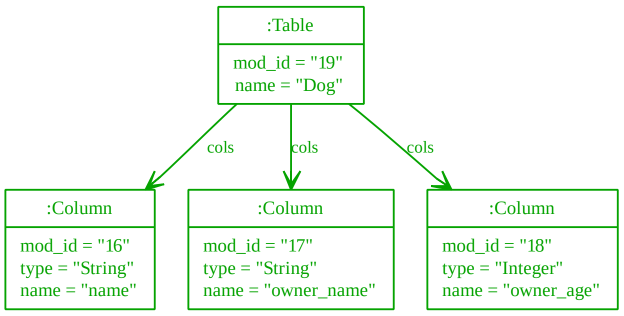


Figure 6: Target model.

formation instances contain two separate lists of equal length related to tracing information. The first list contains tuples (encoded as lists) relating source and target elements. The second list contains the name of the transformation rule which created the corresponding entry in the first list. The fact that they are stored separately is a simple implementation detail — conceptually these two lists can be considered to constitute one single piece of information.

The `TM.Visualizer` module defines a function `visualize_trace(transformation)` which takes a transformation instance and visualizes it complete with tracing information. The result of visualizing the tracing information for the example model of section 4.6 can be seen in figure 7. The original source model is on the left in blue, with the target model on the right in green. The black lines between source and target elements are the traces between source and target elements. Individual traces always run from a single source element to a single target element. Each trace has a name of the form t*n* 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 the 't1' trace from the source class to the target table is a result of the `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. Similarly since two traces share the name 't4', one can determine that a rule – in this case `User_Type_Attr_To_Column` – created more than one target element in a single execution.

Although this subsection has talked about how tracing information is stored and visualized, and what the visualization can be used for, it has not discussed how the tracing information is created. Section 5.2 explains how tracing information is created, and how users can control its creation.

5.1.1. Alternative visualizations

The tracing information in figure 7 is visualized with the source and target models formatted exactly as they were when presented individually in figures 5 and 6. Whilst this visualization works well for small trans-

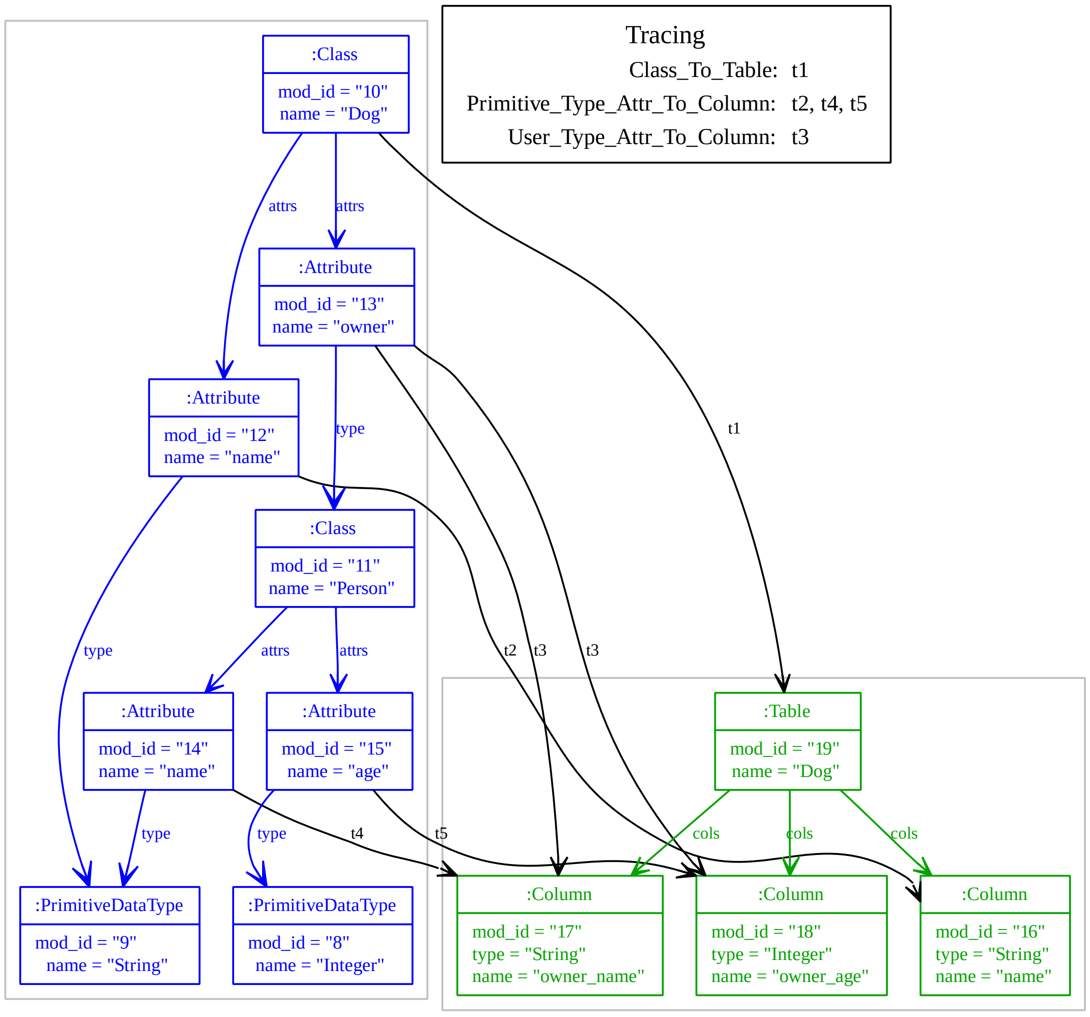


Figure 7: Visualizing tracing information.

formations, larger transformations with greater volumes of tracing information tend to become unreadable as the strict formatting of the source and target models forces traces to overlap with each other. There is thus an alternative form of visualization available via the visualizers `visualize_trace_clustered`[4] function where the source and target elements can be formatted directly alongside one another. Figure 8 shows this alternative visualization. Note that the diagram colouring now becomes critical to distinguish source and target model elements from one another. Due to its general lack of cluttering, this is generally the preferred visualization when tracing information is involved, and is used in the remainder of this paper.

5.2. Standard tracing information creation mechanism

Section 5.1 showed how MT and TM can visualize the tracing information automatically created by MT transformations. In this subsection I outline how the default tracing information is created by MT. Most, if not all, model transformation approaches are currently somewhat vague on this subject. There is therefore little prior art to use as a basis, or point of comparison, for any such mechanism. MT takes a simple approach to the problem to ensure that its behaviour is predictable from a users perspective – this is vital to ensure that

[4]The ‘clustered’ part of this function name reflects the mechanism used in GraphViz to enable this layout.

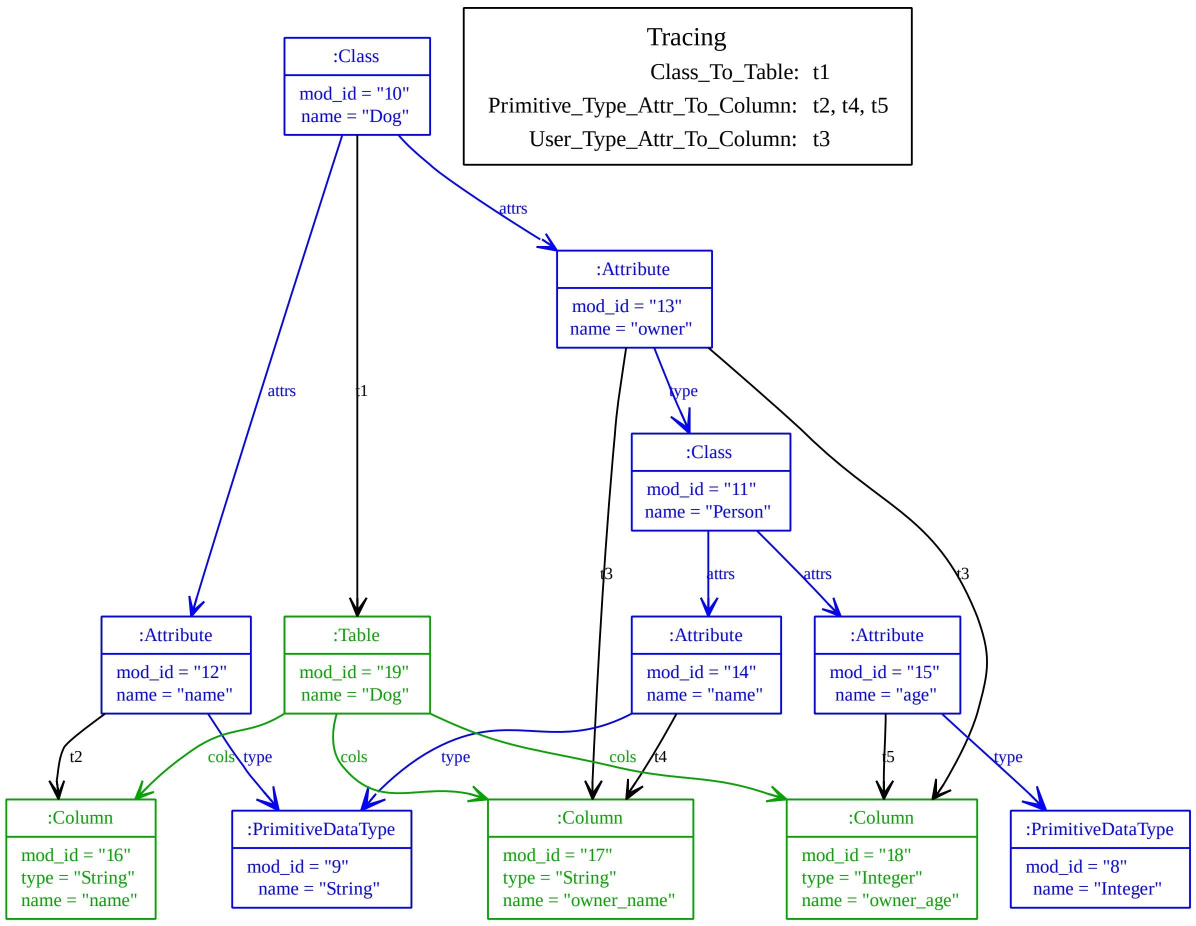


Figure 8: Visualizing tracing information with the free layout of source and target model elements.

users can make informed choices about when and where to add or override tracing information (see section 5.3).

As standard, tracing information in a rule is created between all source elements matched by non-nested model element patterns, and all target elements produced by model element expressions, nested or otherwise. 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 initially seem somewhat arbitrary to try to minimise the source elements used in tracing information whilst maximising 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 as 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. Section 7.16 shows how nested model element patterns can be made to contribute towards tracing information if desired (that section also shows the large number of extra, largely uninteresting, traces created).

The standard tracing information mechanism can be seen in practice by comparing the visualized trace information of figure 8 with the transformation that created it in section 4.5.

5.3. 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. MT provides a simple capability for augmenting, or overriding, the default tracing information created by the standard mechanism.

For example, using the MT example presented in section 4.5 as a base, imagine that one wishes to add extra traces between the source class and all target columns. In order to achieve this one makes use of the optional `tracing_add` clause on MT rules. This clause must contain a single Converge expression which evaluates to a tuple relating source and target model elements. The tuple is then added to the tracing information created automatically by the rule. The new `Class_To_Table` rule looks as follows:

```
rule Class_To_Table:
  srcp:
    (Class, <c>)[name == <n>, attrs == <A>]

  tgtp:
    (Table)[name := n, cols := columns]

  tgt_where:
    columns := []
    for attr := A.iterate():
      columns.extend(self.transform([""], [attr]).flatten())

  tracing_add:
    [[c], columns]
```

Note that since `c` is a single element it needs to be placed within a list to create a valid trace tuple. The tuple in the `tracing_add` clause is then added to the tracing information automatically created by the rule – hence the new traces have the same tracing number (in this case ‘t1’) as the default traces for the rule execution. Figure 9 shows the resulting visualization of the transformation with the extra tracing information added in.

In some circumstances, users may wish to entirely override the default tracing information, rather than simply augmenting it. The `tracing_override` clause in a rule turns off the rules default tracing generation, replacing it with the tuple returned by the single Converge expression in the clause. `tracing_add` and `tracing_override` are thus mutually exclusive clauses within a rule. Whilst maintaining the additional tracing information created by our modified `Class_To_Table` rule, assume one now wishes the other two rules in the transformation to be prevented from generating any tracing information at all. In order to achieve this, `tracing_override` clauses which contain tuples relating the empty set of source elements to the empty list of target elements are defined. The modified rules are as follows:

```
rule User_Type_Attr_To_Column:
  srcp:
    (String, <prefix>)[]
    (Attribute)[name == <n>, type == (Class)[name == <cn>, attrs == <CA>]]
```

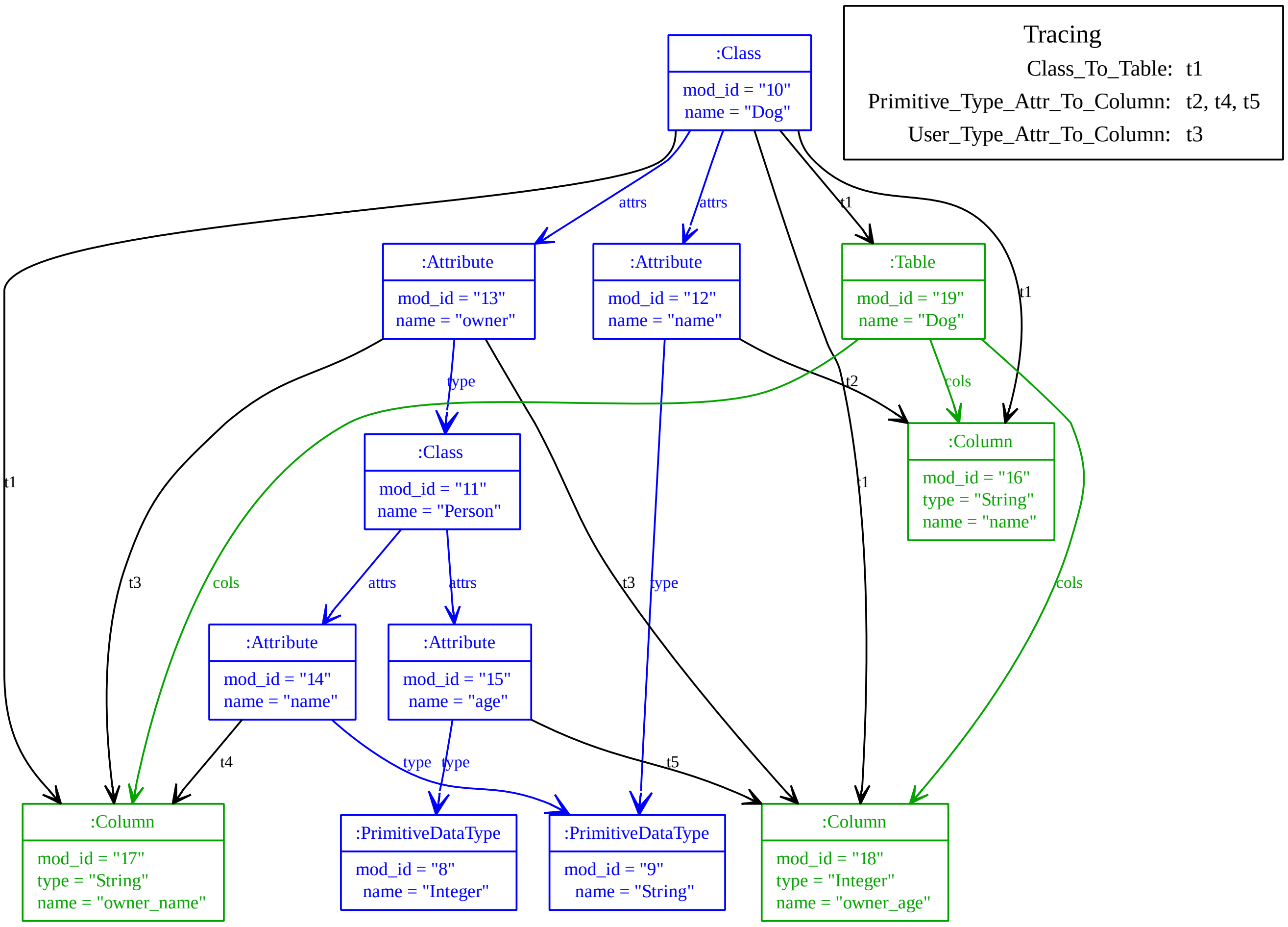


Figure 9: Augmenting the default tracing information.

```
    tgtp:
      self.transform([concat_name(prefix, n)], [ca]) for ca := CA.iterate()

    tracing_override:
      [[], []]

  rule Primitive_Type_Attr_To_Column:
    srcp:
      (String, <prefix>)[]
      (Attribute)[name == <n>, type == (PrimitiveDataType)[name == <pn>]]

    tgtp:
      [(Column)[name := concat_name(prefix, n), type := pn]]

    tracing_override:
      [[], []]
```

The result of running the transformation with its three rules altered can be seen in figure 10. As this example shows, users can completely customise the tracing information created by MT to their own needs.

6. Towards more sophisticated transformations

The previous section introduced the basics of the MT language, via a simple version of the running example. In this section, I delve into some of the more advanced aspects of the MT language which allow more

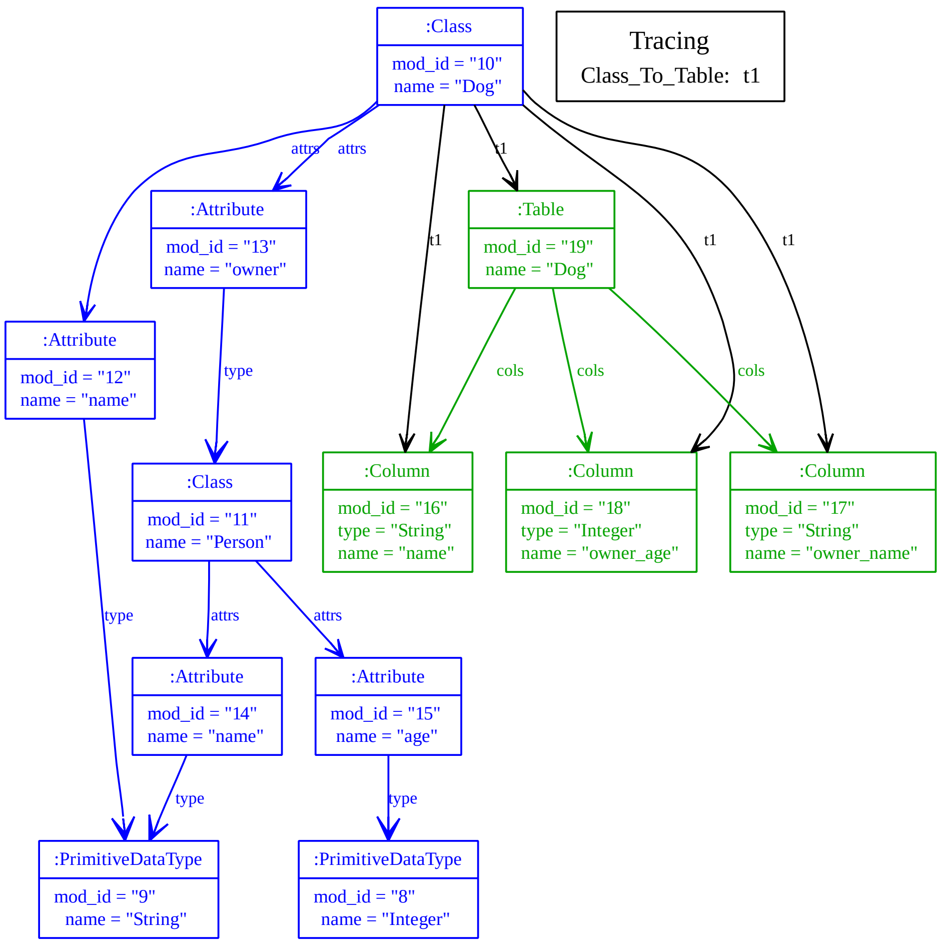


Figure 10: Augmenting and overriding the default tracing information.

complex and sophisticated transformations to be expressed. In order to explore these aspects fully, I first present a more complex version of the running example.

6.1. Extending the running example

In this subsection I define the ‘advanced’ variant of the running example. The overall idea is, as before, to translate UML-esque class models into relational database models. In order to make the example more challenging, the ‘Simple UML’ meta-model is extended in several ways as can be seen in figure 11. These extensions extend the required transformation as follows:

- Associations are added to the meta-model. Associations add a significant degree of complexity to the meta-model because a classes ‘real’ attributes are determined by the union of the attributes it directly links to, and the associations for which it is a source.
- 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.
- Classes which have the `is_persistent` attribute set to true will be converted 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`

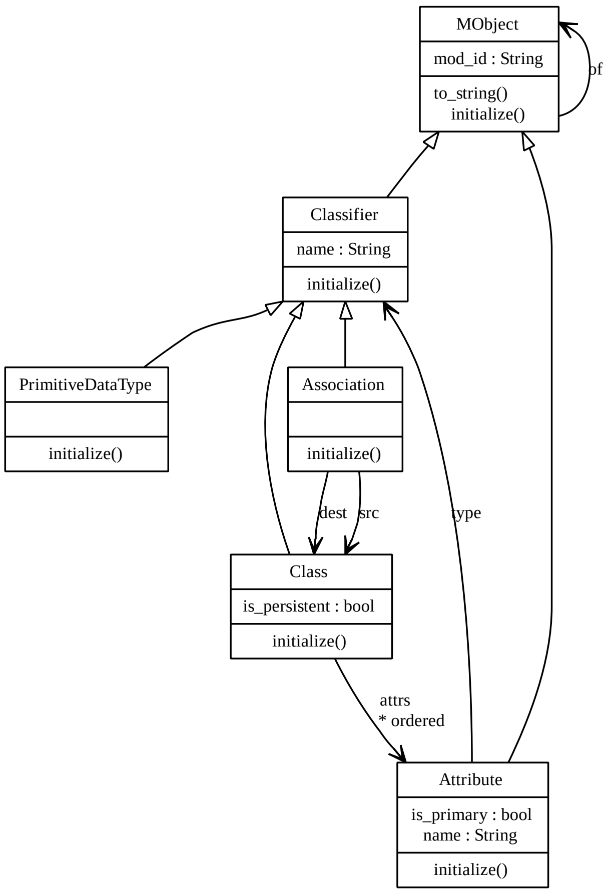


Figure 11: Extended ‘Simple UML’ meta-model.

attribute set to true will not be transformed into tables, and will have their attributes drilled into, as in the simple transformation.

The relational database meta-model is also extended, as shown in figure 12. The extended meta-model allows tables to define primary keys and foreign keys. Note that, since the TM data model allows nested data types to be expressed, foreign keys are defined as a sequence of sequences of columns.

6.2. Pattern multiplicities

One of the problems noted in section 3.4.2 with 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 requires cumbersome work arounds.

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*. Multiplicities specify how often a given source pattern can, or must, match against its source elements. Multiplicities are therefore a constraint on the universe of model elements passed in the parameter corresponding to the patterns position in the `srcp` clause. Each pattern in a `srcp` clause can optionally be suffixed with a multiplicity and

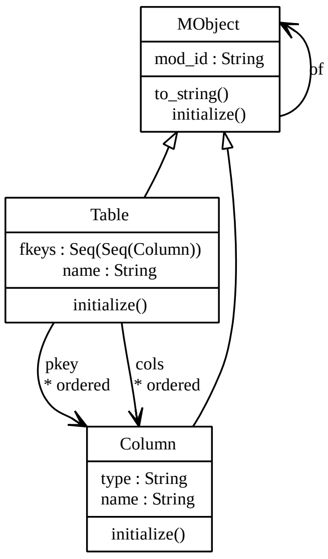


Figure 12: Extended relational database meta-model.

an associated variable binding. The following example of a pattern multiplicity will match zero or more associations, assigning the result of the match to the `assocs` variable:

```
(Association, <assoc>)[name == n] : * <assocs>
```

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

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 `..` *n* `?`	Will match against the minimum number of source elements once `m` elements have been matched, but will not exceed *n* matches.
m `..` `*`	Must match no less than *m* 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 however much simpler in the case of textual regular expressions since text is an inherently ordered data type. Thus the length of matches 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 thus 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.

6.2.1. 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. 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:

```
(Association)[src == (Class)[name = n]] : * <assocs>
(Class, <c>)[name == n]
```

A first glance may suggest that when the rule these patterns are a part of runs, `c` will be set to the class which has the same name as the associations source class. 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 class 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:

```
(Association)[src == (Class)[name = <n>]] : * <assocs>
```

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

```
[Dict{"n" : "orders"}, Dict{"n" : "parts"}]
```

The `MT` module provides a convenience function `mult_extract(bindings, name)` 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 in MT is to use this function with a self variable binding in a model element pattern, which allows a user to determine all the model elements matched by a particular pattern multiplicity.

6.3. Extended example

In this subsection I show the MT version of the extended example. The added complexity in this version of the transformation over the original simpler version is due to three considerations:

1. 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.

2. Classes which are marked as persistent must be transformed substantially different from those not marked as persistent.

3. Foreign keys and primary keys reference columns. It is important that the column model elements pointed to by a table are the appropriate model elements, and not duplicates. Th

The MT 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. Unlike the simple version of the example, there is no need 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.

One feature in particular requires explanation to make sense of this transformation. Many of the rules have more patterns than there are arguments passed to the `transform` function. For example, the `Association_Non_Pe`
rule 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, which . Without such a mechanism, transformations such as this would be complicated by the need to continually pass around the root set of source elements.

The overall structure of this transformation is hopefully relatively straight forward. The `Persistent_Class_To_`
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.

Figure 13 shows a visualization of a particular execution of the transformation. The size of the source model has been increased to the maximum that can be sensibly visualized on paper, to provide some reassurance that MT can cope with transformations beyond a small handful of elements. Note that when freed of paper-based space constraints, and the visualization technique can easily cope with much larger source models.

6.4. Pruning the target model

One thing not immediately obvious from viewing figure 13 is that the final target model is not 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 non-primary key columns). Knowing that, as an implementation detail, TM assigns each new model element a unique and monotonically increasing identifier, one can see from figure 13, that some elements have been discarded, due to the non-contiguous model identifiers in target model elements. For example, the lowest identifier for a target model element is 29 and the highest 47, but identifiers such as 42 are missing in the figure – these are elements that were produced by a rule execution, but discarded by other rules.

MT's approach to achieving the final target model involves firstly taking the model elements produced

Tracing	
Persistent_Class_To_Table:	t1, t12, t19
Primary_Primitive_Type_Attribute_To_Columns:	t2, t4, t5, t13, t14, t16, t20
Persistent_Association_To_Columns:	t3, t15
Association_Non_Persistent_Class_To_Columns:	t6
Non_Primary_Primitive_Type_Attribute_To_Columns:	t7, t8, t9, t10, t11, t17, t18, t21, t22

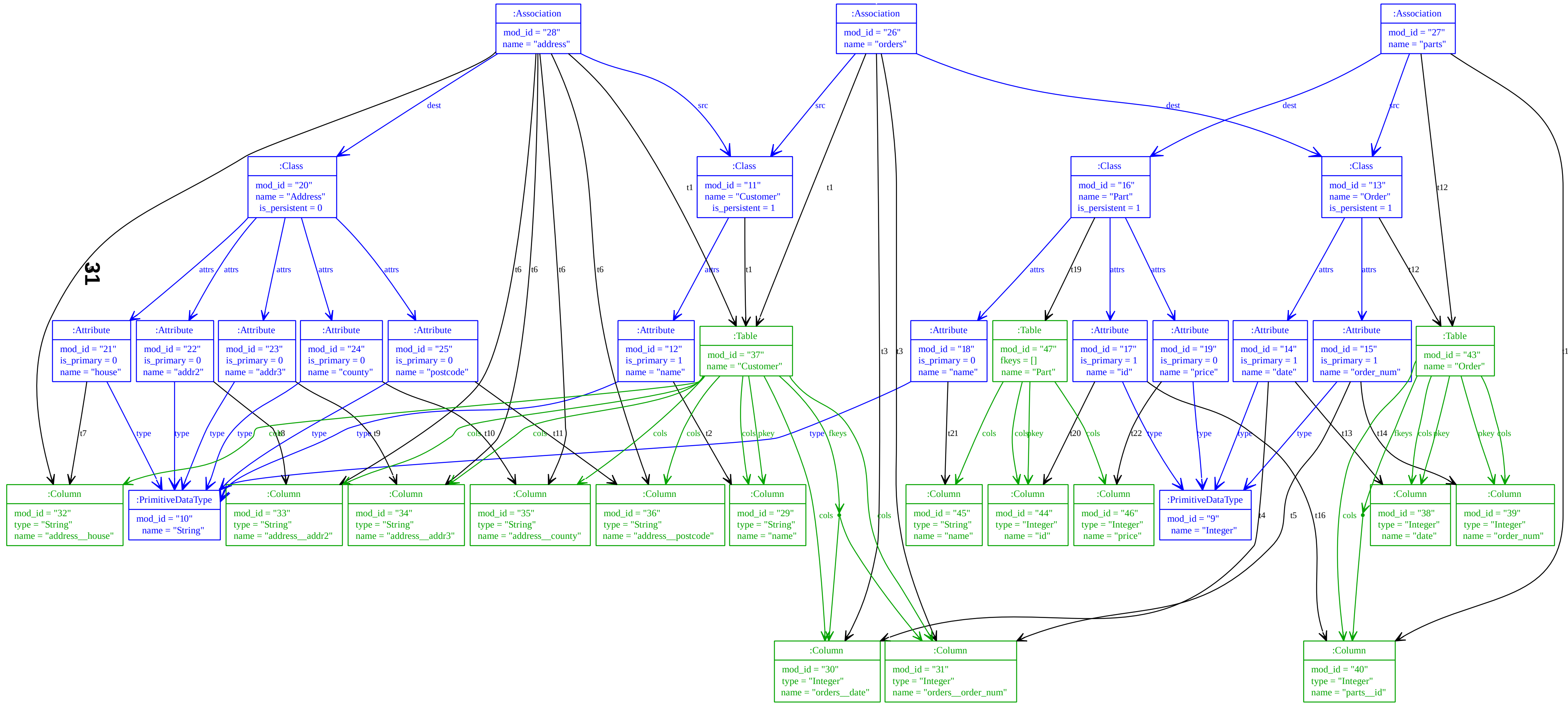

:Association mod_id = "28" name = "address"
:Association mod_id = "26" name = "orders"
:Association mod_id = "27" name = "parts"
:Class mod_id = "20" name = "Address" is_persistent = 0
:Class mod_id = "11" name = "Customer" is_persistent = 1
:Class mod_id = "16" name = "Part" is_persistent = 1
:Class mod_id = "13" name = "Order" is_persistent = 1
:Attribute mod_id = "21" is_primary = 0 name = "house"
:Attribute mod_id = "22" is_primary = 0 name = "addr2"
:Attribute mod_id = "23" is_primary = 0 name = "addr3"
:Attribute mod_id = "24" is_primary = 0 name = "county"
:Attribute mod_id = "25" is_primary = 0 name = "postcode"
:Attribute mod_id = "12" is_primary = 1 name = "name"
:Table mod_id = "37" name = "Customer"
:Attribute mod_id = "18" is_primary = 0 name = "name"
:Table mod_id = "47" fkeys = [] name = "Part"
:Attribute mod_id = "17" is_primary = 1 name = "id"
:Attribute mod_id = "19" is_primary = 0 name = "price"
:Attribute mod_id = "14" is_primary = 1 name = "date"
:Attribute mod_id = "15" is_primary = 1 name = "order_num"
:Table mod_id = "43" name = "Order"
:Column mod_id = "32" type = "String" name = "address__house"
:PrimitiveDataType mod_id = "10" name = "String"
:Column mod_id = "33" type = "String" name = "address__addr2"
:Column mod_id = "34" type = "String" name = "address__addr3"
:Column mod_id = "35" type = "String" name = "address__county"
:Column mod_id = "36" type = "String" name = "address__postcode"
:Column mod_id = "29" type = "String" name = "name"
:Column mod_id = "45" type = "String" name = "name"
:Column mod_id = "44" type = "Integer" name = "id"
:Column mod_id = "46" type = "Integer" name = "price"
:PrimitiveDataType mod_id = "9" name = "Integer"
:Column mod_id = "38" type = "Integer" name = "date"
:Column mod_id = "39" type = "Integer" name = "order_num"
:Column mod_id = "30" type = "Integer" name = "orders__date"
:Column mod_id = "31" type = "Integer" name = "orders__order_num"
:Column mod_id = "40" type = "Integer" name = "parts__id"
dest
src
attrs
type
cols
pkey
fkeys
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
t13
t14
t15
t16
t19
t20
t21
t22

Figure 13: An example execution of the extended transformation.

by transforming each element in the root source set. It then then 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 in the eventual target model. Note that scheme does allow the eventual target model to consist of unconnected subgraphs.

6.5. Combinators

One of the most interesting features in the QVT-Partners approach are combinators. Section 3.3 showed the `or` combinator; the QVT-Partners approach also defines `and` 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 | O}]

  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 sensible 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.

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. I believe a fruitful area of future research will be to investigate more powerful combinators, with a view to including the most useful in a standard library.

7. Implementation

In this section I discuss some of the most interesting aspects of the MT implementation. Since the implementation follows the typical structure of DSL implementation functions – as seen concretely in TM [Tra05] – many aspects of the implementation have already been presented elsewhere. In this section I outline the novel aspects of the implementation, relative to what has already been presented.

7.1. MT Grammar

Since any discussion of MT's implementation necessarily references MT's grammar, I first present the CPK grammar:

```
mt_rules ::= "TRANSFORMATION" "ID" "NEWLINE" mt_rule { "NEWLINE" mt_rule }*

mt_rule ::= "RULE" "ID" ":" "INDENT" mt_in "NEWLINE" mt_out "DEDENT"
```

```
mt_in   ::= mt_inp mt_inc
mt_inp  ::= "SRCP" ":" "INDENT" pt_ipattern { "NEWLINE" pt_ipattern }* "DEDENT"
mt_inc  ::= "NEWLINE" "SRC_WHEN" ":" "INDENT" pt_ipattern "DEDENT"
        ::=

mt_tgt  ::= mt_tgtp mt_tgtw mt_tracing
mt_tgtp ::= "TGTP" ":" "INDENT" mt_tgt_expr { "NEWLINE" expr }* "DEDENT"
mt_tgtw ::= "NEWLINE" "TGT_WHERE" ":" "INDENT" expr { "NEWLINE" expr }* "DEDENT"
        ::=
mt_tracing ::= "NEWLINE" "TRACING_ADD" ":" "INDENT" expr "DEDENT"
           ::= "NEWLINE" "TRACING_OVERRIDE" ":" "INDENT" expr "DEDENT"
           ::=

pt_ipattern ::= pt_ipattern_expr pt_ipattern_qualifier

pt_ipattern_expr ::= pt_iobj_pattern   %precedence 10
                 ::= pt_iset_pattern   %precedence 10
                 ::= pt_ivar           %precedence 10
                 ::= expr

pt_ipattern_qualifier ::= ":" pt_multiplicity "<" "ID" ">"
                      ::=

pt_multiplicity             ::= pt_multiplicity_upper_bound
                            ::= expr "!"
                            ::= "*" "!"
                            ::= expr "." "." pt_multiplicity_upper_bound
pt_multiplicity_upper_bound ::= expr
                            ::= expr "?"
                            ::= "*"
                            ::= "*" "?"

pt_iobj_pattern            ::= "(" pt_iobj_pattern_self ")" "[" pt_iobj_slot
                               pt_iobj_pattern_comparison pt_ipattern_expr { ","
                               pt_iobj_slot pt_iobj_pattern_comparison
                               pt_ipattern_expr }* "]"
                           ::= "(" pt_iobj_pattern_self ")" "[" "]"
pt_iobj_pattern_self       ::= "ID" "," "<" "ID" ">"
                           ::= "ID"
                           ::=
pt_iobj_slot               ::= "ID"
                           ::= "ID" "(" expr { "," expr }* ")"
                           ::= "ID" "(" ")"
pt_iobj_pattern_comparison ::= "=="
                           ::= "!="
                           ::= "<"
                           ::= ">"
                           ::= "<="
                           ::= ">="
               ::= "IS"

pt_iset_pattern       ::= "Set{" pt_iset_pattern_elems "|" pt_iset_pattern_elems "}"
                      ::= "Set{" pt_iset_pattern_elems "}"
pt_iset_pattern_elems ::= pt_ipattern { "," pt_ipattern }*
                      ::=

pt_ivar ::= "<" "ID" ">"

mt_tgt_expr           ::= expr mt_tgt_expr_qualifier
mt_tgt_expr_qualifier ::= "FOR" expr
                      ::=

expr ::= pt_mep_pattern

pt_mep_pattern ::= "(" "ID" ")" "[" "ID" ":=" expr { "," "ID" ":=" expr }* "]"
               ::= "(" "ID" ")" "[" "]"
```

7.2. Outline of the implementation

The translation of an `MT.MT` block into Converge is relatively straight forward at a high-level. An MT transformation is translated to a single class with a number of standard functions (e.g. `transform` and `get_target`, as seen earlier), fields for holding tracing information and so on, and a function for each rule in the transformation. The translation records the names of all transformation rules as a list in the `_rule_names` field within the transformation class; the list retains the rules' order in the source file.

The following subsections show how rules are translated and the definitions of the standard functions.

7.3. Translating rules

The translation of a rule into a function conceptually follows the path outlined by example in section 4.1. The translated function takes a variable number of arguments, each of which must be a list containing the 'universe' of elements which each pattern in the rule can match against. If the translated source model clauses fail to match against the arguments passed to the translated function, the rule fails. If these clauses succeed, they return the set of model elements matched by model element patterns, and a dictionary of bindings. The dictionary of bindings is passed to the translated `tgtp` and `tgtp_where` clauses which produce and return a list of target elements. The matched model element patterns and target elements are then used to create a suitable tuple for the transformations tracing execution, before the rule returns the target elements produced.

A simplified version of the outer translation of a rule is as follows:

```
1   func _t_mt_rule(node):
2     // mt_rule ::= "RULE" "ID" ":" "INDENT" mt_src "NEWLINE" mt_out "DEDENT"
3     return [|
4       bound_func $<<CEI.name(node[2].value)>>(*objs):
5         if not mep_objects, bindings := $<<self.preorder(node[5])>>.apply(objs):
6           return fail
7
8         if not target_elements := $<<self.preorder(node[7])>>(bindings):
9           raise Exceptions.Exception(Strings.format(\${}<<CEI.lift( \
10            "Failed to generate anything for '%s'.")>>, objs.to_str()))
11
12        self._tracing.append([mep_objects, target_elements])
13
14        return target_elements
15    |]
```

Line 5 translates the rules source model clauses; note that if the source model clauses fail, the entire rule fails. If the source model clauses succeed, then the translation of the target model clauses in line 8 is executed. Failure of the target model clauses is deemed a fatal error, and an exception is raised. Note that there is no concept of backtracking between the target and source model clauses – once the source model clause has successfully matched, the target model clauses are executed. The final part of the rule in line 12 creates the necessary tracing information.

The translation of a rules source model clauses contains subtleties to ensure that backtracking amongst patterns works correctly; this is explored in the upcoming subsections. Since the target model clauses are essentially already fairly standard imperative code (with the addition of model element expressions), their translation is largely uninteresting and consequently elided. However, the translation of all clauses is complicated by the need to embed normal Converge code, and to ensure that there are no unintended interactions between translated and embedded code (see sections 7.13 and 7.15).

7.4. Translating a rules source model clauses

A rule potentially contains two source model clauses: the `srcp` and `srcp_when` clauses, the latter of which is optional. Each pattern in the `srcp` clause is translated into a generator function which takes in a list of model elements, and a Converge dictionary of bindings. Each time the pattern matches successfully it returns a list containing three items: a list of the model elements matched by model element patterns, a dictionary of new bindings, and the object the pattern evaluated to. The last of these is largely an internal detail needed to support the nesting of patterns. Section 7.5 contains more detail on the translation of patterns.

Since each pattern is a generator, it needs to be placed within a `for` construct to ensure that possible matches can be generated. When more than one pattern is present in a `srcp` clause, patterns must be translated 'inside out' into nested `for` constructs; that is, the first pattern will be the outermost `for` construct, and the last pattern the innermost. This slightly complicates the translation, which iterates over the patterns in the order they are presented in the parse tree. In order to achieve the desired effect, a standard idiom is used. Patterns are first translated to a temporary list. The translated `srcp_when` clause, if it exists, is used as the innermost construct. The temporary list is then iterated over in reverse order with each iteration placing the result of the previous iteration inside a `for` construct. This idiom is highly useful, and also shows a simple example of a DSL translation where the translated code does not directly reflect the order of its source. Noting that a lists `riterate` function iterates in reverse order over the list, the simplified version of this translation is as follows:

```
translated_patterns := [ordered translated patterns]
patterns_expr := [| $<<self.preorder(src_when clause)>> |]
for translated_pattern := translated_patterns.riterate():
  patterns_expr := [|
    for $<<translated_pattern>>:
      $<<patterns_expr>>
  |]
```

This simple translation only deals with part of the problem caused when the failure of a pattern causes backtracking to an earlier clause. As each pattern matches, it returns a list containing three items: a list of the model elements matched by model element patterns, a dictionary of new bindings, and the object the pattern evaluated to. As each pattern in the `srcp` clause is matched, the rules records of matched model element patterns and variable bindings grow. When a pattern fails, the list of elements it matched by model element patterns and variable bindings it created need to be 'undone' from the rules' records. Since the failure of one pattern may cause the failure of an arbitrary number of preceding patterns, the 'undo' mechanism also needs to work to an arbitrary depth.

MT makes use of Converge's variable capturing and scoping rules to implement a simple and reasonably efficient undoing mechanism. Each rule defines two variables `matched_mp_elems` and `bindings` which store the rules evolving list of matched model pattern elements and variable bindings. These variables are available to each translated pattern. Each translated pattern then defines two variables private to that pattern (hidden via Converge's scoping rules), named `matched_mp_elems_backup` and `bindings_backup`. As the names of these variables may suggest, they are used to store the values of the `matched_mp_elems` and `bindings` variables as they were before the pattern is matched; if the pattern did not match successfully or is required to generate new matches, they are used to restore their value. A slightly elided version of the translation function is as follows:

```
func _t_mt_src(node):
  // mt_src  ::= mt_srcp mt_srcc
  // mt_srcp ::= "SRCP" ":" "INDENT" pt_spattern { "NEWLINE" pt_spattern }* "DEDENT"

  translated_patterns := [ordered translated patterns]
  if node[2].len() > 1:
    // mt_srcc ::= "NEWLINE" "SRC_WHEN" ":" "INDENT" expr "DEDENT"
    patterns_expr := [|
      if $<<self.preorder(node[2][5])>>:
        return [&matched_mp_elems, &bindings]
     |]
  else:
    // mt_srcc ::=
    patterns_expr := [| return [&matched_mp_elems, &bindings] |]

  for i := (translated_patterns.len() - 1).to(-1, -1):
    patterns_expr := [|
      if $<<CEI.lift(i)>> < &args.len():
        elements := &args[$<<CEI.lift(i)>>]
      else:
        elements := &self._root_set

      matched_mp_elems_backup := &matched_mp_elems
      bindings_backup := &bindings
      for new_matched_mp_elems, new_bindings, matched_elem := \
        $<<translated_patterns[i]>>(&bindings, elements):
        &matched_mp_elems := &matched_mp_elems + new_matched_mp_elems
        &bindings := &bindings + new_bindings
        $<<patterns_expr>>
        &matched_mp_elems := matched_mp_elems_backup
        &bindings := bindings_backup
    |]

  return [|
    func (*args):
      if args.len() > $<<CEI.lift(translated_patterns.len())>>:
        return fail

      matched_mp_elems := Set{}
      bindings := Dict{}

      $<<patterns_expr>>

      return fail
  |]
```

7.5. Translating patterns

In this subsection I show how patterns are translated in MT (the translation of pattern multiplicities is detailed in section 7.11). Each pattern is translated into a generator function which takes in a list of model elements, and a Converge dictionary of bindings. Each time the pattern matches successfully it returns a list containing three items: a list of the model elements matched by model element patterns, a dictionary of new bindings, and the object the pattern evaluated to.

Patterns can be any one of a number of pattern expressions: model element patterns, set patterns, variable bindings, and normal Converge expressions. Pattern expressions may arbitrarily nest other pattern expressions. Each pattern expression is translated to a generator which can generate zero or matches against given model elements. The translation is complicated by the fact that nested pattern expressions are also generators; therefore when a pattern is asked by backtracking to generate new matches, the backtracking may need to resume several levels deep in a nested pattern expression.

All translated pattern expressions contain a wrapper which iterates over the objects passed to the pattern and passes them one at a time to the pattern expression. Since the pattern expression is a generator, its result is immediately yielded to the translated patterns caller. Noting that `yield` in Converge is an expression, the outer translation is as follows:

```
func _t_pt_spattern(node):
  // pt_spattern ::= pt_spattern_expr
  return [|
    func (bindings, elements):
      for element := elements.iterate() & yield $<<self.preorder(node[1])>> \
        (bindings, element)

      return fail
  |]
```

In the following subsections I detail how each type of pattern expression is translated by MT.

7.6. Translating variable bindings

As befits the simplest type of pattern expression, MT's translation of variable bindings is simple:

```
1  func _t_pt_svar(node):
2    // pt_svar ::= "<" "ID" ">"
3
4    self._pattern_vars.add(node[2].value)
5    var_str := CEI.lift(node[2].value)
6    return [|
7      func (bindings, element):
8        if bindings.contains($<<var_str>>) & not bindings[$<<var_str>>] == element:
9          return fail
10       return [Set{}, Dict{$<<var_str>> : element}, element]
11   |]
```

Lines 8 and 9 check to see whether the variable in question has already been bound; if it has, the value of the existing binding is compared against the element being matched to ensure equality. If the original and new binding values are not equal, the variable binding fails. As such, this behaviour is largely redundant in MT since exactly the same effect can be achieved by having an initial variable binding followed by references to that variable. This behaviour is maintained for the sake of ensuring backwards compatibility with the QVT-Partners approach.

As the MT translation encounters variable bindings, it adds them to the set of known variable bindings in the translations' `_pattern_vars` field (line 4). This information is used in two different ways. Firstly the set of variable bindings is used to determine the valid variable references for subsequent patterns and clauses in a rule; this is necessary since variable bindings in patterns with multiplicities are dealt with differently (see sections 6.2 and 7.11). Secondly variables in a Converge expression which reference a variable binding need to be translated into a dictionary lookup on the current set of known bindings (see section 7.13).

7.7. Translating model element patterns

The translation of model patterns is the largest individual part of MT's translation, but can be split into two parts: matching the model element type and dealing with the self variable; matching against model element slots. The former part is relatively simple. The latter part is complicated by the need to deal with nested pattern expressions. In this subsection I first present a simplified translation of model element patterns which does not deal with nested pattern expressions, before presenting the complete translation.

7.7.1. A simplified translation

In order to understand the translation of model element patterns, I first consider a simplified variant. The model element patterns in this simple variant can not contain any reference to the self variable, and slot comparisons with nested pattern expressions will only be evaluated once. If a slot comparison fails, the entire pattern fails immediately. In the interests of brevity, only equality and inequality slot comparisons are translated. This subset still encompasses a number of interesting and useful model element patterns such as the following:

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

The simplified translation is as follows:

```
func _t_pt_smodel_pattern(node):
  // pt_smodel_pattern ::= "(" pt_smodel_pattern_self ")" "[" pt_sobj_slot
  //                        pt_smodel_pattern_comparison pt_spattern_expr { ","
  //                        pt_sobj_slot pt_smodel_pattern_comparison
  //                        pt_spattern_expr }* "]"
  //                   ::= "(" pt_smodel_pattern_self ")" "[" "]"

  // pt_sobj_pattern_self ::= "ID"
  type_match := [|
    if not TM.type_match($<<CEI.lift(node[2][1].value)>>, &element):
      return fail
  |]

  slot_comparisons := []
  while i < node.len() & node[i].conforms_to(List) & node[i][0] == "pt_sobj_slot":
    slot_name := node[i]
    slot_comparison := node[i + 1]
    slot_pattern := node[i + 2]

    if slot_comparison[1].type == "==":
      slot_condition := CEI.ieq_comparison
    elif slot_comparison[1].type == "!=":
      slot_condition := CEI.ineq_comparison

    slot_comparisons.append([|
      slot_element := &element.$<<CEI.name(slot_name[1].value)>>
      if not new_matched_mep_elems, new_bindings, matched_elem := \
        $<<self.preorder(slot_pattern)>>(bindings, &slot_element):
        return fail
      if not $<<slot_condition([| &slot_element |], [| matched_elem |])>>:
        return fail
      &local_bindings += new_bindings
    |])

    i += 4

  return [|
    func (bindings, element):

      local_bindings := Dict{}

      $<<type_match>>

      $<<slot_comparisons>>

      return [Set{element}, local_bindings, element]
  |]
```

Lines 9 to 12 deal with ensuring the model element to be matched is of the correct type. Lines 25 to 33 shown the heart of the translation of slot comparisons. Line 26 extracts the value of the model elements slot. Line 27 evaluates the slots pattern; if the slots pattern fails for any reason, the entire model pattern fails. The hitherto unused third element, hitherto referred to as ‘the object the pattern evaluated to’, returned by the pattern expression is then compared to the value of the model elements slot obtained in line 26, using the slot comparison operator. If the comparison operator fails, then the entire model pattern fails. As can be seen in line 47, a model element pattern ignores any model elements matched by nested model element patterns.

The clumsy term ‘the object the pattern evaluated to’ is used because conceptually there are two distinct ways in which a pattern expression evaluates. Converge expressions used as patterns (e.g. `"Fred"`; see section 7.9) simply evaluate to an object which must be checked against the model elements slot. However other types of pattern expressions, such as the model element pattern `(Person)[name != "Fred"]`, are passed the value of the model elements slot and asked to match against it; they then return the value of the model elements slot unchanged. In other words, some types of pattern expressions (Converge expressions) evaluate to a new object whilst some return the object passed to them (e.g. model element patterns). Note that even in the latter case it is necessary to check the slot comparison after the evaluation, so that element patterns such as the following (which is functionally equivalent to the previous example) evaluate correctly:

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

7.7.2. Ensuring the complete evaluation of nested pattern expressions

The preceding translation of model element patterns contains one major flaw: it does not correctly deal with nested pattern expressions that may generate more than one match. Consider the following pattern:

```
(Dog)[name == <n>, allowable_foods == Set{<x> | <Y>}]
```

The set pattern `Set{<x> | <Y>}` will potentially generate a match for every element in a set matched against it. If, for example, the rule this pattern is part of contains a `src_when` clause along the lines of `x == "Biscuit"` then it is vital that the set pattern generates all possible matches to ensure that a correct match can be found (if one exists). In the previous translation of model elements, unless the nested set pattern happened to stumble across the correct combination during its first iteration, then the entire rule this is a part of would fail.

In the general case, pattern expressions may be nested to an arbitrary depth within one another. MT thus needs to ensure that all pattern expressions, no matter how deep they are nested, can generate all their possible matches. For model element patterns, it is also desirable that the pattern expressions in slot comparisons generate multiple matches in a predictable fashion. Pattern expressions are thus evaluated in a deliberately similar fashion to patterns in a `srcp` clause in the order that they were defined, from left to right. If a model element pattern is requested to generate more matches, the right most pattern expression will generate all a further match if possible. When a pattern expression within a model element pattern generates all its possible matches, the pattern expression to its left generates a new match, which causes the control flow to return to its right, causing that pattern to generate a new match. When all pattern expressions within a model element pattern have generated all their matches, then the model element pattern itself fails. In common with patterns in a `srcp` clause, MT ensures that each time a pattern expression fails, the appropriate variable bindings are ‘undone’.

In order to cope with arbitrary levels of nested pattern expressions, one might reasonably expect a significant degree of complexity to be needed in the translation – indexes within lists needing to be passed around and stored, and so on. However by careful use of Converge generators and the conjunction operator, the desired effect can be achieved with a relatively small amount of code. Considering the same marginally simplified variant of model element patterns as previously (references to the self variable not allowed; only a subset of slot comparison operators dealt with), the translation is as follows:

```
1  func _t_pt_smodel_pattern(node):
2    type_match := [|
3      if not TM.type_match(\$<<CEI.lift(node[2][1].value)>>, &element):
4        return fail
5    |]
6
7    returns_vars := []
8    current_bindings_var := CEI.ivar(CEI.fresh_name())
9    conjunction := [[| $<<current_bindings_var>> := &bindings |]]
10
11   while i < node.len() & node[i].conforms_to(List) & node[i][0] == "pt_sobj_slot":
12     slot_name := node[i]
13     slot_comparison := node[i + 1]
14     slot_pattern := node[i + 2]
15
16     if slot_comparison[1].type == "==":
17       slot_condition := CEI.ieq_comparison
18     elif slot_comparison[1].type == "!=":
19       slot_condition := CEI.ineq_comparison
20
21     next_bindings_var := CEI.ivar(CEI.fresh_name())
22     return_var := CEI.ivar(CEI.fresh_name())
23     returns_vars.append(return_var)
24     conjunction.append([| $<<return_var>> := $<<func_>>($<<current_bindings_var>>) \
25       |])
26     conjunction.append([| $<<next_bindings_var>> := $<<current_bindings_var>> + \
27       $<<return_var>>[1] |])
28     current_bindings_var := next_bindings_var
29     i += 4
30
31   conjunction.append([| [Set{&element}, Functional.foldl(_adder, \
32     Functional.map(_element1, $<<CEI.ilist(returns_vars)>>)), &element] |])
33
34   return [|
35     func (bindings, element):
36
37       $<<type_match>>
38
39       for yield $<<CEI.iconjunction(conjunction)>>
40
41       return fail
42   |]
43
44 func _adder(x, y):
45   return x + y
46
47 func _element0(x):
48   return x[0]
49
50 func _element1(x):
51   return x[1]
```

Note that in this code `_adder`, `_element0` and `_element1` are module level functions. These functions are used in later translations in this paper.

The underlying theme in this translation is that pattern expressions in slot comparisons may be generators. Pattern expressions are placed into a single conjunction expression (chiefly built up in lines 24 and 25). The translation places the conjunction containing translated pattern expressions within a `for` construct (line 38), which yields a value each time a successful match of all slot comparisons is found. Thus the translation utilizes Converge's built-in goal-directed evaluation to ensure that all possible values – including those from nested pattern expressions – for all translated slot comparisons are evaluated.

There are two further (somewhat related) aspects of the translation which require explanation. The first of these relates to MT's treatment of variable bindings, particularly the need to 'undo' variable bindings when a slot comparison fails and Converge backtracks. Essentially after each slot comparison has been added to the conjunction, MT creates a new uniquely named variable (line 21) which has assigned to it the union of the existing variable bindings and those created by the pattern expression (lines 25). The pattern expression in the next slot comparison then uses this union of variable bindings as its set of currently valid bindings (line 24). When Converge backtracks, the currently valid bindings are implicitly undone since the union of existing and new bindings is performed after each translated slot comparisons.

The second aspect relates to the value returned by a model element pattern, which is created in lines 31 to 32. Since model element patterns ignore elements matched by nested model element patterns (section 5.2), it is not surprising that the first element of the returned list is a set containing only the element matched by the current model element pattern. The second element of the returned list which is initially rather foreboding using as it does the `foldl` and `map` functions which operate as their LISP counterparts. Before tackling it directly, we first need to investigate the `return_vars` variable in the translation, which is a list containing quasi-quoted variables. Essentially each time a nested pattern expression is evaluated, a new `return_var` variable is created (line 22) to which the return value of the pattern expression is assigned (line 24). The `return_var` variable is then added to the `return_vars` list (line 23). Each `return_var` variable thus holds a standard three element list. The `foldl` call in line 31 is then passed a list of lists at run-time; each sub-list will be a list containing three elements, as returned by a pattern expression. The `_element1` function then selects the variable bindings generated from each slot comparison; the `_adder` function then creates a union of these variable bindings. This union is a non-strict subset of the final value of, `current_bindings_var` which will include all the bindings passed to the model element pattern in its `bindings` argument. Note that whilst it may initially appear simpler to make `return_vars` a run-time variable to which each pattern expressions return list is appended, this would then lead to complications when back-tracking would require items in the list to be removed. However since the variables in `return_vars` are known at compile-time, it would be possible to achieve a small optimization by moving the `foldl` call to compile-time; this is left as an exercise for the reader.

It is interesting to compare this translation to that used for patterns in a `srcp` clause, as presented in section 7.4. While the two transformations are essentially functionally equivalent, the earlier translation is perhaps more initially appealing since it reuses a familiar concept (nested `for` constructs). Although the translation in this section uses the less familiar conjunction operator, it results in a much shorter, more idiomatic – and marginally more efficient – translation. As this may suggest, making use of some of the less common features present in Converge can be of significant advantage when translating DSLs.

7.8. Translating set patterns

In this subsection I outline the translation of set patterns, but do not delve into the code of the translation which uses the same techniques and idioms outlined in the translation of model element patterns.

Essentially set patterns match against single element patterns (those to the left of the '`|`' character) and subset patterns (those to the right of the '`|`' character) simultaneously. For each single element pattern, MT iterates over the set being matched; no two single element patterns will be matched against the same element simultaneously. For each subset pattern, MT iterates over the powerset of the set to match against. The intersection of all subsets (including the set comprised of all single element patterns matched) must be $\emptyset$. The union of all subsets (including the set comprised of all single element patterns matched) must equal the set being matched. Set patterns then generate an appropriate return value whenever each single element pattern and each subset pattern match successfully.

7.9. Translating Converge expressions when used as patterns

As outlined in section 7.7.1, when Converge expressions are used as pattern expressions, they act in a different fashion to other pattern expressions. Whereas all other types of pattern expressions are a declarative match against model elements, Converge expressions are simply expected to evaluate to constants (in this situation meaning integers, strings, model elements and so on). MT therefore defines that Converge expressions in this situation are only evaluated once – even if the particular Converge expression is a generator, it will only ever be required to generate a single value. Converge expressions used as pattern expressions return a list consisting of the empty set to represent the model elements matched by model element patterns, an empty dictionary of bindings, and the constant object the expression evaluated to.

The translation of Converge expressions in this instance thus requires only a very thin wrapping around the actual expression itself:

```
func _t_pt_spattern_expr(node):
  // pt_spattern_expr ::= pt_sobj_pattern
  //                  ::= pt_sset_pattern
  //                  ::= pt_svar
  //                  ::= expr

  if node[1][0] == "expr":
    return [|
      func (bindings, elements):
        return [Set{}, Dict{}, $<<self.preorder(node[1])>>]
    |]
  else:
    return self.preorder(node[1])
```

Section 7.13 details how the Converge grammar rule `expr` is embedded in the MT grammar.

7.10. An example translated pattern

Having now seen the translations of variable bindings, model element patterns, set patterns and Converge expressions used as pattern expressions, we are now in a position to see the result of translating a particular pattern. I use the following pattern, which incorporates all three types of pattern expressions:

```
(Dog)[name == <n>, allowable_foods == Set{"pork" | <Y>}]
```

The result of translating this pattern is the following ITree:

```
unbound_func (bindings, element){
  if not Input_Pattern_Creator.TM.type_match("Dog", element):
    return Input_Pattern_Creator.fail
  for yield $$76$$ := bindings & $$78$$ := unbound_func (bindings){
    slot_element := element.name
    for matched_mp_elems, new_bindings, matched_elem := unbound_func (bindings, \
      element){
      if bindings.contains("n") & not bindings["n"] == element:
        return Input_Pattern_Creator.fail
      return [Set{}, Dict{"n" : element}, element]
    }(bindings, slot_element):
      if slot_element == matched_elem:
        yield [matched_mp_elems, new_bindings, matched_elem]
    return Input_Pattern_Creator.fail
  }($$76$$) & $$77$$ := $$76$$ + $$78$$[1] & $$84$$ := unbound_func (bindings){
    slot_element := element.allowable_foods
    for matched_mp_elems, new_bindings, matched_elem := unbound_func (bindings, \
18       element){
19       if not element.conforms_to(Input_Pattern_Creator.Set):
20         return Input_Pattern_Creator.fail
21       if element.len() < 1:
22         return Input_Pattern_Creator.fail
23       for $$79$$ := element.iterate() & $$80$$ := unbound_func (bindings, elements){
24         return [Set{}, Dict{}, "pork"]
25       }(bindings, $$79$$) & $$81$$ := Input_Pattern_Creator.Functional. \
26         powerset_generator(element) & $$81$$.union(Set{$$79$$}) == element & not \
27         $$81$$.contains($$79$$) & $$82$$ := unbound_func (bindings, element){
28         if bindings.contains("Y") & not bindings["Y"] == element:
29           return Input_Pattern_Creator.fail
30         return [Set{}, Dict{"Y" : element}, element]
31       }(bindings, $$81$$) & $$82$$[2] == $$81$$:
32         yield [$$80$$[0] + $$82$$[0], $$80$$[1] + $$82$$[1], element]
33       return Input_Pattern_Creator.fail
34     }(bindings, slot_element):
35       if slot_element == matched_elem:
36         yield [matched_mp_elems, new_bindings, matched_elem]
37     return Input_Pattern_Creator.fail
38   }($$77$$) & $$83$$ := $$77$$ + $$84$$[1] & [Set{element}, Input_Pattern_Creator. \
39     Functional.foldl(Input_Pattern_Creator._adder, Input_Pattern_Creator.Functional. \
40     map(Input_Pattern_Creator._element1, [$$78$$, $$84$$])), element]
41   return Input_Pattern_Creator.fail
42 }(bindings, element)
```

Despite its initial appearance as an impenetrable jumble of bizarrely named identifiers, through careful examination of the input pattern, and the translations presented in this section, it is possible to identify which parts of this ITree relate to specific parts of the input pattern. The first step in this is to recursively break the input pattern down into its constituent pattern expressions. One can then determine which line numbers each pattern expression relates to. A simple table showing this is as follows (note that due to the recursive breakdown, outer pattern expressions line numbers overlap with those of nested pattern expressions):

Pattern	Lines
`(Dog)[name == <n>, allowable_foods == Set{"pork" | <Y>}]`	1 – 42
`<n>`	6 – 11
`Set{"pork" | <Y>}`	20 – 34
`"pork"`	23 – 25
`<Y>`	27 – 31

7.11. Translating pattern multiplicities

In order to deal with patterns with multiplicities (see section 6.2), some additions need to be made to the outer translation of patterns from section 7.5. Although each of the several forms of pattern multiplicities requires a specific translation, they all follow the same general form, which can be split into two distinct phases. In order to demonstrate this, I present the translation for the `*` multiplicity in an elided view of the `_t_pt_spattern` function:

```
func _t_pt_spattern(node):
  // pt_spattern ::= pt_spattern_expr pt_spattern_qualifier
  // pt_spattern_qualifier ::= ":" pt_multiplicity "<" "ID" ">"
  // pt_multiplicity ::= pt_multiplicity_upper_bound
  // pt_multiplicity_upper_bound ::= "*"

  self._inside_multiplicity_pattern += 1
  pattern := [|
    func (bindings, elements):
      matches := []
      for element := elements.iterate() & matches.append($<<self.preorder( \
12          node[1])>>(bindings, element))
13
14        powerset := Functional.powerset(matches)
15        powerset := Sort.sort(powerset, func (x, y) {
16          return x.len() < y.len()
17        })
18        for matches := powerset.riterate():
19          if matches.len() == 0:
20            continue
21          yield [Functional.foldl(_adder, Functional.map(_element0, matches)), \
22            Dict{$<<CEI.lift(node[2][4].value)>> : Functional.map(_element1, \
23            matches)}, Functional.foldl(adder, Functional.map(func (x) {
24            return [x[2]]
25          }, matches))]
26
27        return [Set{}, Dict{$<<CEI.lift(node[2][4].value)>> : []}, elements]
28    |]
29    self._inside_multiplicity_pattern -= 1
30
31    return pattern
```

The first phase of a multiplicities execution involves matching elements. In the case of the `*` multiplicity, this occurs in lines 10 to 12 which evaluates every successful match of the pattern (note that when the pattern does not match successfully, backtracking ensures that the `append` call will not be executed). The second phase of execution then successively returns permutations of the matches. Note that, although not the case for the `*` multiplicity, in some multiplicities these two phases will be partially intertwined. Lines 14 to 17 evaluate the powerset[5] of matches, sorting the resulting permutations into ascending order based on the number of elements in each. The `for` construct in line 18 then iterates over the the powerset in reverse order, yielding permutations of lesser size as the multiplicity is called upon to generate new matches. The list yielded by the multiplicity in lines 21 to 25 is simpler than it may first appear. Line 21 unions the elements matched by model element patterns from each match in the permutation. Lines 22 and 23 create a single binding for the multiplicities variable, assigning it a list of variable bindings, with a bindings entry for each match in the permutation. Lines 23 to 25 union the objects each match in the permutation evaluated to.

The translation of pattern multiplicities requires a small but important change to the translation of variable bindings, to prevent variable bindings within multiplicities from being added to the `_pattern_vars` field. The `_inside_multiplicity_pattern` field within the translation tracks whether the translation is currently processing a pattern multiplicity or not. The updated `_t_pt_svar` function thus looks as follows:

```
func _t_pt_svar(node):
  // pt_svar ::= "<" "ID" ">"

  if _inside_multiplicity_pattern == 0:
    self._pattern_vars.add(node[2].value)
  var_str := CEI.lift(node[2].value)
  return [|
    func (bindings, element):
      if bindings.contains($<<var_str>>) & not bindings[$<<var_str>>] == element:
        return fail
      return [Set{}, Dict{$<<var_str>> : element}, element]
  |]
```

[5]The Converge `powerset` function returns a list of lists if, as in this case, it is passed a list rather than a set.

7.12. Standard functions

Each transformation has two standard functions of particular importance: the `transform` and `transform_all` functions. In this subsection I show the definition of these two functions.

The `transform` function was outlined in section 4.1. It takes a variable number of arguments, each of which must be a list. Noting that function objects in Converge have a function `apply` which takes a list of values and applies them to the function as if they were passed as individual arguments, the `transform` function looks as follows:

```
1  func transform(*elems):
2    for elem := elems.iterate():
3      if not elem.conforms_to(List):
4        raise Exceptions.Type_Exception(List, elem.instance_of, elem.to_str())
5
6    for rule_name := self._rule_names.iterate():
7      if target := self.get_slot(rule_name).apply(elems):
8        return target
9
10   raise Exceptions.Exception(Strings.format("Unable to transform ’%s’.", elems.to_str()))
```

The first action of the `transform` function After is to type-check its arguments in lines 2 to 4. It then makes use of the `_rule_names` field within a transformation which records the names of a transformations rule in the order they were defined. Iterating over the `_rule_names` field allows the translated rule function to be accessed via the `get_slot` function. Using this, the `transform` function calls rules in the order in which they were defined, succeeding as soon as it finds a rule which executes on the input. If no rules execute, an exception is raised.

The `transform_all` function is a simple, but highly useful, convenience function built on top of the `transform` function. Given a list of model elements, it transforms each using the `transform` function. The definition of `transform_all` is thus simple:

```
func transform_all(elems):
  if not elems.conforms_to(List):
    raise Exceptions.Type_Exception(List, elems.instance_of, elems.to_str())

  target_elems := []
  for elem := elems.iterate():
    target_elems.append(self.transform([elem]))

  return target_elems
```

7.13. Embedding Converge code within DSLs

When compared to other model transformation approaches, one of MT’s most novel aspects is its ability to embed GPL code within it. This is possible due to Converge’s DSL embedding features. The ability to embed Converge code in DSL’s benefits both the DSL’s users and implementers. Users can reuse their knowledge of standard Converge, whilst DSL implementers can reuse tried and trusted parts of the Converge compiler. In this subsection I explain how a DSL’s can embed Converge within itself.

The key to embedding normal Converge code can be seen in such as `pt_ipattern_expr` in the MT grammar (section 7.1) which reference the `expr` rule from the Converge grammar (appendix A). The first point to note is that all rules in the MT grammar are prefixed by `mt_`; this allows the MT grammar to be merged with the Converge grammar with no conflicts. Since CPK grammars are currently defined in strings, merging two grammars together is simply a case of adding two strings. There is currently no notion of grammar namespaces nor are any checks for conflicts between the two grammars. As this may suggest, whilst the current implementation of this feature is workable, it is one of the less refined parts of DSL implementation in Converge.

Once the DSL's grammar has been merged with the main Converge grammar, the next question is how to have the Converge grammar handle parse tree fragments related to the Converge grammar? Currently the translation class in MT subclasses the `IModule_Generator` class from the Converge compiler. `IModule_Generator` contains the translation for each rule in the Converge grammar and works in exactly the same way as a DSL translation class. Thus the MT subclass only need add appropriate traversal functions for rules specific to the MT grammar. Although subclassing of large and complex classes is often thought of as being dangerous, the `IModule_Generator` module has been specifically designed with sub-classing of this sort in mind. Normal Converge expressions are thus translated into ITree's 'for free' by the `IModule_Generator` module, and with little interference from MT. The MT subclass in fact requires only one interaction with its superclass, needing to override the `_t_var` function. Essentially references to variable bindings are translated into dictionary lookups on the `bindings` variable (see section 7.6); see also section 7.15. An elided version of the `_t_var` translation function is as follows:

```
func _t_var(node):
  // var ::= "ID"

  if self._pattern_vars.contains(node[1].value):
    return [| &bindings[$<<CEI.lift(node[1].value)>>] |]
  else:
    return exbi IModule_Generator._IModule_Generator._t_var(node)
```

In summary – despite the need to add strings representing grammars together, and to subclass a complex class residing in the depths of the Converge compiler – the process of embedding Converge code in DSLs is surprisingly simple and relatively free of complications. However it is unclear whether this approach would scale satisfactorily to larger examples. I believe that in the future two things may need to be changed to improve the situation. Firstly grammars need to be properly modularised to ensure that naming problems between grammars do not arise, and that the relationship between grammars is clearly stated. Secondly it would be useful to loosen the coupling between DSLs and the `IModule_Generator` module, possibly by removing the requirement to subclass the `IModule_Generator` class.

7.14. Extending the Converge grammar

Although this section has thus far ignored the translation of a rules target clauses, the presence of model element expressions in such clauses is worthy of examination. As a brief recap, model element expressions such as `(Dog)[name := "Fido", owner := (Person)[name = "Fred"]]` create new model elements; they are syntactically similar, although not identical, to model element patterns. Section 4.4 contains more details on model element expressions. Model element expressions can be used anywhere in a rules target clauses that a normal Converge expression can be used. Although this may suggest that model element expressions can only be used at the top-level within target clauses, they can in fact be used within Converge expressions themselves. For example the following expression shows how a model element expression can be used within a Converge list:

```
[(Person)[name = "Fred"]]
```

As this example shows, in the context of target clauses, model element expressions effectively embed themselves in the the base Converge language itself.

The embedding of model element expressions is currently implemented by taking advantage of the fact that that CPK grammars are strings, and that production rules can have alternatives added at any point in the grammar. Thus in the MT grammar (section 7.1) the `expr` rule from the Converge grammar is extended with a new alternative by MT pointing to the `pt_mep_pattern` rule. Since the `expr` translation function in the Converge compiler immediately hands computation over to the rule named in its alternatives, the MT translation class needs only to provide a simple translation function for `pt_mep_pattern`.

It should be noted that whilst extremely powerful, this technique is not generally applicable. It currently requires detailed knowledge of the Converge grammar and the Converge compilers internals in order to ensure that extending a rule in the Converge grammar has the desired effect. I hope that future versions of Converge will be able to provide safer support for extension of this sort.

7.15. Preventing unintended interaction between translated and embedded code

One of the challenges not tackled in the TM DSL was preventing unintended variable capture from DSL input and the translated DSL code. This problem arises when an ITree derived from user input is placed inside an ITree containing dynamically scoped variables. As seen in the translations in this section, dynamically scoped variables occur frequently, chiefly via the `&var` syntax. Dynamically scoped variables are highly useful in allowing ITree's to be built piece meal. However whilst statically scoped variables are automatically safely renamed by Converge's scoping rules dynamically scoped variables may cause variable capture with ITree's derived from user input. For example, the variable `bindings` is frequently dynamically scoped in the translations of this section; if an MT were to use the same variable name in, for example, a `tgt_where` clause, then unexpected results would almost certainly arise.

To prevent this problem occurring, MT performs its own α-renaming of variables in Converge expressions. MT takes advantage of the fact that each ITree can report its free and bound variables (via the `get_free_vars` and `get_bound_vars` functions respectively). Essentially for each rule, MT first calculates the free and bound variables of all Converge expressions contained in that rules clauses. For each variable, a fresh name is then generated; a dictionary records the mapping between the original and fresh names. When MT encounters Converge variables during its translation, it translates them to variables with the corresponding fresh name. By renaming all variables from users input, MT thus ensures that there can be no unintended variable capture.

Once all variables have been safely renamed, a rules free variables then require extra treatment. For example the `concat_name` function in section 4.5 is a free variable in the context of the `Classes_To_Tables` transformation, since it is defined outside of the transformation. All instances of the `concat_name` within a given rule will be renamed to a variable along the lines of `$$5$concat_name$$`. At this point, there is no link between the value of `concat_name` outside the rule, and the value of `$$5$concat_name$$`. Thus MT adds to translated rules assignments from the original value of variables to their fresh name equivalent. In the case of the `concat_name` function, the result of the translation would look along the lines of the following:

```
$$5$concat_name$$ := concat_name

...
  $$5$concat_name$$("", bindings["n"])
...
```

7.15.1. Assigning to variables in outer blocks

Although the α-renaming mechanism of variables in user input prevents unintended variable capture, it introduces problems due to the disconnect between the original variable and its fresh-named clone. This leads to two related problems.

The first problem relates to assigning to free variables. Since a fresh-named clone is made of each free variable, assigning to free variables in an MT block does not affect the value of the original variable. MT thus mirrors the normal Converge expectation that variables assigned to in a block (where a block in MT is essentially a rule) are local to that block. However a problem arises if one wishes an assignment to a free variable. First, let us assume that MT allows some free variables to be declared as nonlocal (recall that in normal Converge `nonlocal x` is a declaration that assignment to the variable *x* does not create a local *x*, but instead binds to the the first outer block which contains an assignment of *x*). Assignment to a free variable then becomes problematic, since the user will be assigning to the variables fresh-named clone; furthermore there is no way to assign to the original variable without reintroducing the prospect of variable capture. A partial solution to this problem is for MT to mirror the assignment of free variables to their fresh-named clone at the beginning of the translated rule, with the assignment of the fresh-named clone to its free variable equivalent at the end of the transformation. Whilst this is possible, it means that during the execution of the rule the local and global values of the variable may differ.

This then highlights the more general problem, which is that at any point during the execution of a rule the values of the original variable and its fresh-named clone may diverge either through assignment to the original variable or its fresh-named clone. There is no solution for this problem at the moment in Converge. Although one can devise increasingly sophisticated work arounds which reduce the potential for the problem to arise, fundamentally, the cloning of variables is flawed since there is no mechanism for atomically synchronising the cloned and original variables.

This therefore highlights a deficiency in Converge. A possible solution to this deficiency would be for Converge to acquire a 'variable alias' feature which would alias a variable `x` in an outer block to `y` in an inner block. Since the names are merely aliases for the same underlying variable, there can then no synchronization issues between the two. Such a feature would ideally work in much the same way as the `nonlocal` declaration; indeed, it is also implicit that aliased variables are nonlocal to the block in which they are renamed. Although the Converge VM provides sufficient support for such a feature, the compiler and language have yet to be sufficiently extended.

7.16. Generating tracing information from nested model patterns

In section 5.2, the standard MT tracing information creation mechanism was outlined. By default, only non-nested model element patterns contribute to the source part of trace tuples. I asserted that empirically this appeared to be a sensible compromise that created sufficient tracing information without overwhelming the user. However it is clear that this technique may not be suitable for all applications; one can easily imagine further research to determine the most practical tracing information creation techniques for different types of transformations. To this end, in this subsection I present a simple modification to the MT translation which changes the default tracing information created by allowing nested model element patterns to contribute to the source part of trace tuples. This serves two separate purposes. Firstly it provides evidence that the default tracing information creation mechanism achieves a useful balance in terms of the volume of information it creates. Second it shows that DSL implementations in Converge tend to be amenable to changes, and also that the MT implementation itself can serve as a testbed for further model transformation experimentation.

The modification to MT necessary to allow nested model expression patterns to contribute to the source part of trace tuples is in fact rather simple. Essentially all that is needed is for model element patterns to return the union of all elements matched by nested model element patterns. By default, model element patterns only return the element they matched against, essentially ignoring the elements matched by nested model element patterns. However the required information is present in the `return_var` associated with each slot comparison in a model element pattern. Thus all that is needed is to use the same technique used to union the bindings of each slot comparison. Replacing lines 31 to 32 of the complete translation from section 7.7 with the following achieves the desired effect:

```
conjunction.append([| [Set{&element} + Functional.foldl(_adder, \
  Functional.map(_element0, $<<CEI.ilist(returns_vars)>>)), \
  Functional.foldl(_adder, Functional.map(_element1, $<<CEI.ilist(returns_vars)>>)), \
  &element] |])
```

Taking exactly the same source model and transformation used in figure 13, the result of making this change to MT can be seen in figure mt:tracing information from nested model pattern expressions. Note that in this new visualization, one can see that many target elements have tracing information from more than one source element; the end result is rather harder to read than figure 13, and does not add significantly to the users understanding of the transformation in this particular case.

7.17. Summary of the implementation

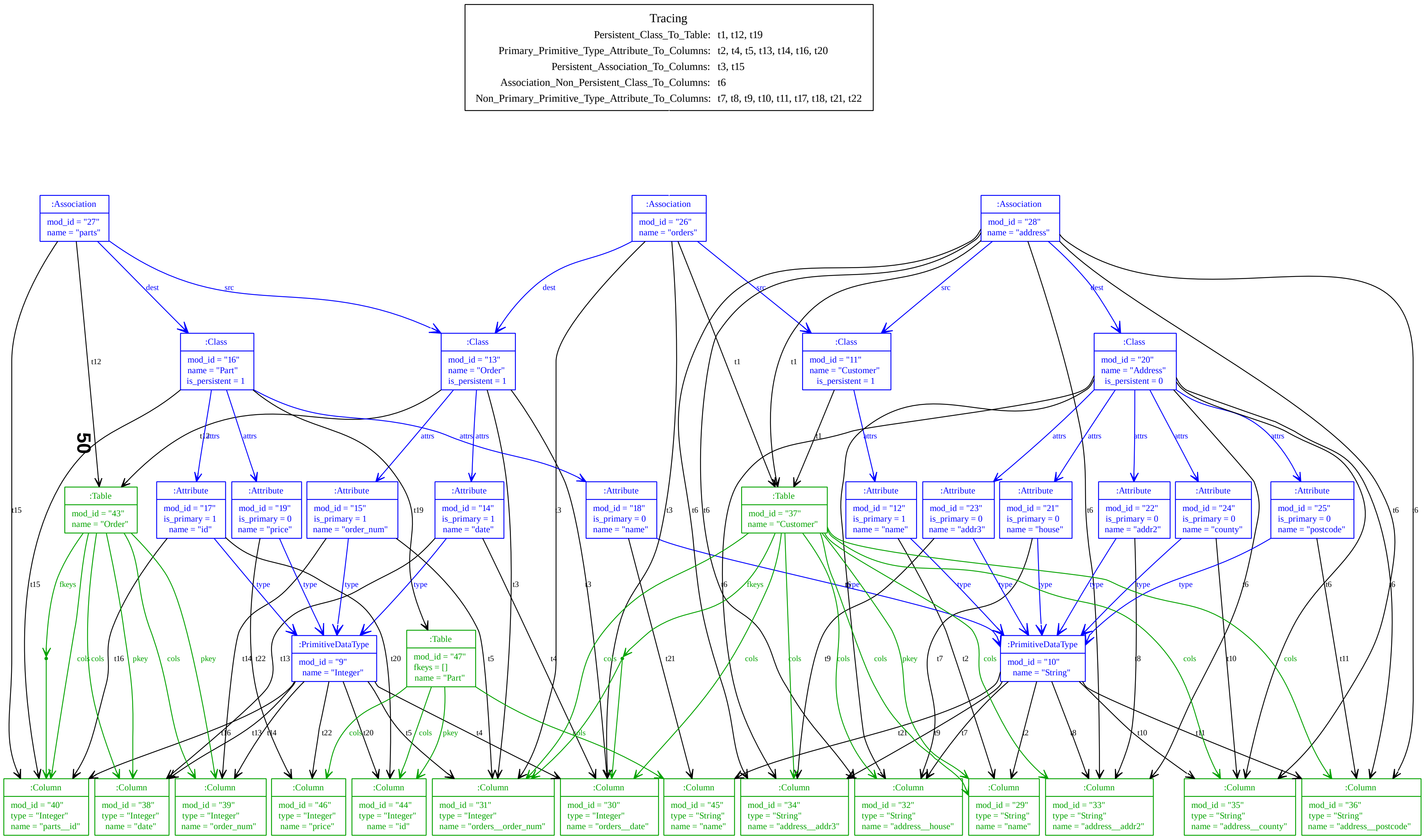
In this section I have presented an analysis of the major parts of the MT implementation. To demonstrate the result of MT's translation of a transformation, section B shows the complete result of translating the simple MT transformation from section 4.5.

8. Related work

As with the majority of existing systems, MT is a unidirectional stateless model transformation system. MT's most obvious ancestor is the QVT-Partners approach [QVT03b] which pioneered the use of patterns in model transformations. MT takes the base QVT-Partners pattern language and enriches it with features such as pattern multiplicities, and variable slot comparisons. 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 3.4, this variant language suffers from several conceptual and practical problems. I believe that MT is unique in being able to embed a GPL within it. Perhaps more significant than the actual language embedded within MT transformations is the ability to call out naturally to normal Converge code, even if it is defined outside of the transformation. 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, it differentiates MT from virtually all existing model transformation approaches, which typically present a highly constrained execution environment.

Perhaps the closest model transformation approach is the commercial XMap language [CESW04], an approach essentially based on the QVT-Partners approach. This also means that the issues noted in both this section, and in section 3.4 with respect to the QVT-Partners approach, apply equally to XMap. XMap is however notable for its sister language XSync which allows changes to be propagated in the style outlined by Tratt and Clark [TC03]. A later publication will show how MT can be evolved into a powerful change propagating language.



Tracing

Persistent_Class_To_Table:	t1, t12, t19
Primary_Primitive_Type_Attribute_To_Columns:	t2, t4, t5, t13, t14, t16, t20
Persistent_Association_To_Columns:	t3, t15
Association_Non_Persistent_Class_To_Columns:	t6
Non_Primary_Primitive_Type_Attribute_To_Columns:	t7, t8, t9, t10, t11, t17, t18, t21, t22

:Association mod_id = "27" name = "parts"
:Association mod_id = "26" name = "orders"
:Association mod_id = "28" name = "address"
:Class mod_id = "16" name = "Part" is_persistent = 1
:Class mod_id = "13" name = "Order" is_persistent = 1
:Class mod_id = "11" name = "Customer" is_persistent = 1
:Class mod_id = "20" name = "Address" is_persistent = 0
:Table mod_id = "43" name = "Order"
:Attribute mod_id = "17" is_primary = 1 name = "id"
:Attribute mod_id = "19" is_primary = 0 name = "price"
:Attribute mod_id = "15" is_primary = 1 name = "order_num"
:Attribute mod_id = "14" is_primary = 1 name = "date"
:Attribute mod_id = "18" is_primary = 0 name = "name"
:Table mod_id = "37" name = "Customer"
:Attribute mod_id = "12" is_primary = 1 name = "name"
:Attribute mod_id = "23" is_primary = 0 name = "addr3"
:Attribute mod_id = "21" is_primary = 0 name = "house"
:Attribute mod_id = "22" is_primary = 0 name = "addr2"
:Attribute mod_id = "24" is_primary = 0 name = "county"
:Attribute mod_id = "25" is_primary = 0 name = "postcode"
:PrimitiveDataType mod_id = "9" name = "Integer"
:Table mod_id = "47" fkeys = [] name = "Part"
:PrimitiveDataType mod_id = "10" name = "String"
:Column mod_id = "40" type = "Integer" name = "parts__id"
:Column mod_id = "38" type = "Integer" name = "date"
:Column mod_id = "39" type = "Integer" name = "order_num"
:Column mod_id = "46" type = "Integer" name = "price"
:Column mod_id = "44" type = "Integer" name = "id"
:Column mod_id = "31" type = "Integer" name = "orders__order_num"
:Column mod_id = "30" type = "Integer" name = "orders__date"
:Column mod_id = "45" type = "String" name = "name"
:Column mod_id = "34" type = "String" name = "address__addr3"
:Column mod_id = "32" type = "String" name = "address__house"
:Column mod_id = "29" type = "String" name = "name"
:Column mod_id = "33" type = "String" name = "address__addr2"
:Column mod_id = "35" type = "String" name = "address__county"
:Column mod_id = "36" type = "String" name = "address__postcode"
dest
src
attrs
type
cols
pkey
fkeys

Figure 14: Tracing information from nested model pattern expressions.

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 [DIC03], 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.

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 9 outlines the work that may resolve some of these limitations.

9. 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, is with regards to 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'. Similarly at the moment all rules exist at the same level; that is, given an element to transform, rules are tried in order. Complex transformations will require more selective mechanisms to determine which rules can be executed, both for structuring efficiency reasons. At the moment, a transformations execution time has a worst case proportional to n^2 where n is the number of rules in the transformation; authors of larger transformations may require that their domain knowledge is used to narrow down the number of rules used to transform given elements. I believe that analysing work on combinators in functional languages may lead to new insights on how to better structure transformations[6].

The desirable for scalability is a concrete manifestation of a more nebulous problem surrounding model transformations: their usability. Whilst one can present advanced tools to users, it is vital that the tools be relatively easy to use. I believe there is significant work to be done in presenting model transformation languages to different users. MT could serve a useful purpose here in allowing model transformation languages to be easily tailored for different audiences.

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.

[6]This suggestion is partly a result of a conversation with Bernhard Rumpe, made during a visit to the Technische Universität Braunschweig in February 2005.

10. Summary

In this paper I presented the MT model transformation language. I started by examining the QVT-Partners approach, from which MT is partly derived, in depth. Identifying the strengths and weaknesses of this approach explains some of the underlying design decisions taken with MT. I then explored MT's basic features, including its novel visualization abilities of transformations, including automatically generated tracing information. I then explored some of MT's more advanced features such as pattern multiplicities, which allowed a sophisticated model transformation to be concisely expressed. I then finished the paper by examining in depth the translation of an MT transformation into MT.

I believe that MT is the first model transformation approach to present a detailed analysis of its implementation. In so doing, I hope that MT demonstrates practical idioms for implementing model transformation engines. This analysis may serve as a useful reference point for model transformation engine authors. As the source code for MT is freely available, I also hope that the analysis will allow others to take MT and alter it for their own purposes. In this way, I hope that MT aids further experimentation into differing model transformation techniques.

MT's implementation is also notable for its relative brevity. Through careful use of Converge idioms such as generators and the use of goal-directed evaluation, I assert that much of the tedious machinery that would be needed if MT were to be implemented in a standard GPL has been avoided. Although it is outside of the scope of this paper to present hard numbers to back up this claim, I believe that MT provides compelling evidence that the seemingly disparate influences on Converge (such as Icon's goal-directed evaluation, ObjVLisp's data model, and Template Haskell's compile-time meta-programming) coalesce to form a natural and highly powerful development environment.

A. Converge grammar

This section lists the CPK grammar for Converge. This is extracted directly from the Converge compiler file `Compiler/CV_Parser.cv`:

```
top_level ::= definition { "NEWLINE" definition }*
          ::=

definition  ::= class_def
            ::= func_def
            ::= import
            ::= var { "," var }* ":=" expr
            ::= splice

import      ::= "IMPORT" dotted_name import_as { "," dotted_name import_as }*
dotted_name ::= "ID" { "." "ID" }*
import_as   ::= "AS" "ID"
            ::=

class_def       ::= "CLASS" class_name class_supers class_metaclass ":" "INDENT"
                    class_fields "DEDENT"
class_name      ::= "ID"
                ::= splice
class_supers    ::= "(" expr { "," expr }* ")"
                ::=
class_metaclass ::= "METACLASS" expr
                ::=
class_fields    ::= class_field { "NEWLINE" class_field }*
class_field     ::= class_def
                ::= func_def
                ::= var ":=" expr
                ::= splice
                ::= "PASS"

func_def           ::= func_type func_name "(" func_params ")" ":" "INDENT"
                       func_nonlocals expr_body "DEDENT"
                   ::= func_type func_name "(" func_params ")" "{" "INDENT"
                       func_nonlocals expr_body "DEDENT" "NEWLINE" "}"
func_type          ::= "FUNC"
                   ::= "BOUND_FUNC"
                   ::= "UNBOUND_FUNC"
func_name          ::= "ID"
                   ::= "+"
                   ::= "-"
                   ::= "/"
                   ::= "*"
                   ::= "<"
                   ::= ">"
                   ::= "=="
                   ::= "!="
                   ::= ">="
                   ::= "<="
                   ::= splice
                   ::=
func_params        ::= func_params_elems "," func_varargs
                   ::= func_params_elems
                   ::= func_varargs
                   ::=
func_params_elems  ::= var func_param_default { "," var func_param_default }*
                   ::= splice
func_param_default ::= ":=" expr
                   ::=
func_varargs       ::= "*" var
                   ::= splice
func_nonlocals     ::= "NONLOCAL" "ID" { "," "ID" }* "NEWLINE"
                   ::=

expr_body ::= expr { "NEWLINE" expr }*

expr ::= class_def
```

```
     ::= func_def
     ::= while
     ::= if
     ::= for
     ::= try
     ::= number
     ::= var
     ::= dict
     ::= set
     ::= list
     ::= dict
     ::= string
     ::= slot_lookup     %precedence 50
     ::= list
     ::= application     %precedence 40
     ::= lookup          %precedence 40
     ::= slice           %precedence 40
     ::= exbi
     ::= return
     ::= yield
     ::= raise
     ::= assert
     ::= break
     ::= continue
     ::= conjunction     %precedence 10
     ::= alternation     %precedence 10
     ::= assignment      %precedence 15
     ::= not             %precedence 17
     ::= neg             %precedence 35
     ::= binary          %precedence 30
     ::= comparison      %precedence 20
     ::= pass
     ::= import
     ::= splice          %precedence 100
     ::= quasi_quotes
     ::= brackets

if      ::= "IF" expr ":" "INDENT" expr_body "DEDENT" { if_elif }* if_else
        ::= "IF" expr "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}" { if_elif }*
            if_else
if_elif ::= "NEWLINE" "ELIF" expr ":" "INDENT" expr_body "DEDENT"
        ::= "NEWLINE" "ELIF" expr "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}"
if_else ::= "NEWLINE" "ELSE" ":" "INDENT" expr_body "DEDENT"
        ::= "NEWLINE" "ELSE" "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}"
        ::=

while ::= "WHILE" expr ":" "INDENT" expr_body "DEDENT" exhausted broken
      ::= "WHILE" expr

for ::= "FOR" expr ":" "INDENT" expr_body "DEDENT" exhausted broken
    ::= "FOR" expr

try           ::= "TRY" ":" "INDENT" expr_body "DEDENT" { try_catch }* try_else
try_catch     ::= "NEWLINE" "CATCH" expr try_catch_var ":" "INDENT" expr_body
                  "DEDENT"
try_catch_var ::= "INTO" var
              ::=
try_else      ::= "NEWLINE" "ELSE" ":" "INDENT" expr_body "DEDENT"
              ::=

exhausted ::= "NEWLINE" "EXHAUSTED" ":" "INDENT" expr_body "DEDENT"
          ::=

broken ::= "NEWLINE" "BROKEN" ":" "INDENT" expr_body "DEDENT"
       ::=

number ::= "INT"

var ::= "ID"
    ::= "&" "ID"
    ::= splice
```

```
string ::= "STRING"

slot_lookup ::= expr "." "ID"
            ::= expr "." splice

list ::= "[" expr { "," expr }* "]"
     ::= "[" "]"

dict ::= "DICT{" expr ":" expr { "," expr ":" expr }* "}"
     ::= "DICT{" "}"

set ::= "SET{" expr { "," expr }* "}"
    ::= "SET{" "}"

application ::= expr "(" expr { "," expr }* ")"
            ::= expr "(" ")"

lookup ::= expr "[" expr "]"

slice ::= expr "[" expr ":" expr "]"
      ::= expr "[" ":" expr "]"
      ::= expr "[" expr ":" "]"
      ::= expr "[" ":" "]"

exbi ::= "EXBI" expr "." "ID"

return ::= "RETURN" expr
       ::= "RETURN"

yield ::= "YIELD" expr

raise ::= "RAISE" expr

assert ::= "ASSERT" expr

break ::= "BREAK"

continue ::= "CONTINUE"

conjunction ::= expr "&" expr { "&" expr }*

alternation ::= expr "|" expr { "|" expr }*

assignment        ::= assignment_target { "," assignment_target }* assignment_type
                      expr
assignment_target ::= var
                  ::= slot_lookup
                  ::= lookup
                  ::= slice
assignment_type   ::= ":="
                  ::= "*="
                  ::= "/="
                  ::= "+="
                  ::= "-="

not ::= "NOT" expr

neg ::= "-" expr

binary    ::= expr binary_op expr
binary_op ::= "*"       %precedence 40
          ::= "/"       %precedence 30
          ::= "%"       %precedence 30
          ::= "+"       %precedence 20
          ::= "-"       %precedence 20

comparison    ::= expr comparison_op expr
comparison_op ::= "IS"
              ::= "=="
              ::= "!="
```

```
             ::= "<="
             ::= ">="
             ::= "<"
             ::= ">"

pass ::= "PASS"

splice       ::= expr_splice
             ::= block_splice
expr_splice  ::= "$" "<" "<" expr ">" ">"
block_splice ::= "$" "<" expr ">" ":" "INDENT" "JUMBO" "DEDENT"

quasi_quotes      ::= expr_quasi_quotes
                  ::= defn_quasi_quotes
expr_quasi_quotes ::= "[|" "INDENT" expr { "NEWLINE" expr }* "DEDENT"
                      "NEWLINE" "|]"
                  ::= "[|" expr { "NEWLINE" expr }* "|]"
defn_quasi_quotes ::= "[D|" definition { "NEWLINE" definition }* "|]"
                  ::= "[D|" "INDENT" definition { "NEWLINE" definition }*
                      "DEDENT" "NEWLINE" "|]"

brackets ::= "(" expr ")"
```

B. Simple classes to tables transformation

```
Classes_To_Tables := class Classes_To_Tables:
  _rule_names := ["Class_To_Table", "User_Type_Attr_To_Column", \
    "Primitive_Type_Attr_To_Column"]
  bound_func init(*root_set){
    self._root_set := root_set
    self._transformed_cache := Dict{}
    self._matched_objs := Set{}
    self._tracing := []
    self._tracing_rule := []
    for rule_name := self._rule_names.iterate():
      self._transformed_cache[rule_name] := Dict{}
    self._output := self.transform_all(root_set)
  }
  bound_func get_source(){
    return self._root_set
  }
  bound_func get_target(){
    return self._output
  }
  bound_func get_conflict_objects(){
    return []
  }
  bound_func transform(*objs){
    for obj := objs.iterate():
      if not obj.conforms_to(MT.List):
        raise MT.Exceptions.Type_Exception(MT.List, obj.instance_of, \
          obj.to_str())
    for rule_name := self._rule_names.iterate():
      if output := self.get_slot(rule_name).apply(objs):
        return output
    raise MT.Exceptions.Exception(MT.Strings.format( \
      "Unable to transform ’%s’.", objs.to_str()))
  }
  bound_func transform_all(objs){
    if objs.conforms_to(MT.List):
      output_objs := []
      for obj := objs.iterate():
        output_objs.append(self.transform([obj]))
    elif objs.conforms_to(MT.Set):
      output_objs := Set{}
      for obj := objs.iterate():
        output_objs.add(self.transform([obj]))
    else:
      raise MT.Exceptions.Exception(objs.instance_of.name)
  return output_objs
}
bound_func Class_To_Table(*objs){
  $$18$$self$$ := self
  if matched_objs, bindings := unbound_func (*args){
    if args.len() > 1:
      return Input_Pattern_Creator.fail
    matched_mp_elems := Set{}
    bindings := Dict{}
    if 0 < args.len():
      $$7$$elements$$ := args[0]
    else:
      $$7$$elements$$ := self._root_set
    $$8$$matched_mp_elems_backup$$ := matched_mp_elems
    $$9$$bindings_backup$$ := bindings
    for $$10$$new_matched_mp_elems$$, $$11$$new_bindings$$, \
      $$12$$matched_elem$$ := unbound_func (bindings, elements){
      for element := elements.iterate() & yield unbound_func (bindings, \
        element){
        if not Input_Pattern_Creator.TM.type_match("Class", element):
          return Input_Pattern_Creator.fail
        for yield $$2$$ := unbound_func (bindings, element){
          if bindings.contains("c") & not bindings["c"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"c" : element}, element]
        }(bindings, element) & $$1$$ := bindings + $$2$$[1] & \
          $$4$$ := unbound_func (bindings){
          slot_element := element.name
          for matched_mp_elems, new_bindings, \
            matched_elem := unbound_func (bindings, element){
            if bindings.contains("n") & not bindings["n"] == element:
              return Input_Pattern_Creator.fail
            return [Set{}, Dict{"n" : element}, element]
          }(bindings, slot_element):
            if slot_element == matched_elem:
              yield [matched_mp_elems, new_bindings, matched_elem]
          return Input_Pattern_Creator.fail
        }($$1$$) & $$3$$ := $$1$$ + $$4$$[1] & $$6$$ := \
          unbound_func (bindings){
          slot_element := element.attrs
          for matched_mp_elems, new_bindings, matched_elem := \
            unbound_func (bindings, element){
            if bindings.contains("A") & not bindings["A"] == element:
              return Input_Pattern_Creator.fail
            return [Set{}, Dict{"A" : element}, element]
          }(bindings, slot_element):
            if slot_element == matched_elem:
              yield [matched_mp_elems, new_bindings, matched_elem]
          return Input_Pattern_Creator.fail
        }($$3$$) & $$5$$ := $$3$$ + $$6$$[1] & [Set{element}, \
          Input_Pattern_Creator.Functional.foldl( \
            Input_Pattern_Creator._adder, \
            Input_Pattern_Creator.Functional.map( \
              Input_Pattern_Creator._element1, [$$2$$, $$4$$, $$6$$])), \
          element]
        return Input_Pattern_Creator.fail
      }(bindings, element)
      return Input_Pattern_Creator.fail
    }(bindings, $$7$$elements$$):
      matched_mp_elems := matched_mp_elems + $$10$$new_matched_mp_elems$$
      bindings := bindings + $$11$$new_bindings$$
      return [matched_mp_elems, bindings]
      matched_mp_elems := $$8$$matched_mp_elems_backup$$
      bindings := $$9$$bindings_backup$$
    return Input_Pattern_Creator.fail
  }.apply(objs):
    self._matched_objs.extend(matched_objs)
    if not rtn := unbound_func (matched_objs, bindings){
      concatted_id := Output_Pattern_Creator.concat_id(matched_objs)
      if self._transformed_cache["Class_To_Table"].contains(concatted_id):
        return self._transformed_cache["Class_To_Table"][concatted_id]
      tracing_i := self._tracing.len()
      $$13$$c$$ := bindings["c"]
      $$14$$n$$ := bindings["n"]
      $$15$$A$$ := bindings["A"]
      $$16$$columns$$ := []
      for $$17$$attr$$ := $$15$$A$$.iterate():
        $$16$$columns$$.extend($$18$$self$$.transform([""], \
          [$$17$$attr$$]).flatten())
      out_elems := []
      out_elems.append(unbound_func (){
        user_args := Dict{"name" : $$14$$n$$, "cols" : $$16$$columns$$}
        all_args := []
        args_processed := 0
        for attr_name := Output_Pattern_Creator.TM.all_attrs_in_order( \
          Output_Pattern_Creator.TM._CLASSES_REPOSITORY["Table"]). \
            iterate():
          if args_processed == user_args.len():
            break
          if user_args.contains(attr_name):
            all_args.append(user_args[attr_name])
            args_processed += 1
          else:
            all_args.append(Output_Pattern_Creator.null)
        return Output_Pattern_Creator.TM._CLASSES_REPOSITORY["Table"]. \
          new.apply(all_args)
      }())
      out_elem := out_elems[0]
      self._transformed_cache["Class_To_Table"][concatted_id] := out_elem
      if not out_elem is Output_Pattern_Creator.null:
        tracing := [Output_Pattern_Creator.List(matched_objs), out_elems]
        tracing := Output_Pattern_Creator.trace_reduce(tracing)
        self._tracing.insert(tracing_i, tracing)
        self._tracing_rule.insert(tracing_i, "Class_To_Table")
      return out_elem
    }(matched_objs, bindings):
      raise MT.Exceptions.Exception(MT.Strings.format("Output pattern of" +
      " Class_To_Table failed to generate anything for ’%s’.", \
      objs.to_str()))
    return rtn
  else:
    return MT.fail
}
bound_func User_Type_Attr_To_Column(*objs){
  $$48$$concat_name$$ := concat_name
  $$49$$self$$ := self
  if matched_objs, bindings := unbound_func (*args){
    if args.len() > 2:
      return Input_Pattern_Creator.fail
    matched_mp_elems := Set{}
    bindings := Dict{}
    if 0 < args.len():
      $$37$$elements$$ := args[0]
    else:
      $$37$$elements$$ := self._root_set
    $$38$$matched_mp_elems_backup$$ := matched_mp_elems
    $$39$$bindings_backup$$ := bindings
    for $$40$$new_matched_mp_elems$$, $$41$$new_bindings$$, \
      $$42$$matched_elem$$ := unbound_func (bindings, elements){
      for element := elements.iterate() & yield \
        unbound_func (bindings, element){
        if not Input_Pattern_Creator.TM.type_match("String", element):
          return Input_Pattern_Creator.fail
        for yield $$20$$ := unbound_func (bindings, element){
          if bindings.contains("prefix") & \
            not bindings["prefix"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"prefix" : element}, element]
        }(bindings, element) & $$19$$ := bindings + $$20$$[1] & \
          [Set{element}, Input_Pattern_Creator.Functional.foldl( \
            Input_Pattern_Creator._adder, \
          Input_Pattern_Creator.Functional.map( \
            Input_Pattern_Creator._element1, [$$20$$])), element]
        return Input_Pattern_Creator.fail
      }(bindings, element)
      return Input_Pattern_Creator.fail
    }(bindings, $$37$$elements$$):
      matched_mp_elems := matched_mp_elems + $$40$$new_matched_mp_elems$$
      bindings := bindings + $$41$$new_bindings$$
      if 1 < args.len():
        $$31$$elements$$ := args[1]
      else:
        $$31$$elements$$ := self._root_set
      $$32$$matched_mp_elems_backup$$ := matched_mp_elems
      $$33$$bindings_backup$$ := bindings
      for $$34$$new_matched_mp_elems$$, $$35$$new_bindings$$, \
        $$36$$matched_elem$$ := unbound_func (bindings, elements){
        for element := elements.iterate() & yield \
          unbound_func (bindings, element){
          if not Input_Pattern_Creator.TM.type_match("Attribute", \
            element):
            return Input_Pattern_Creator.fail
          for yield $$21$$ := bindings & $$23$$ := \
            unbound_func (bindings){
            slot_element := element.name
            for matched_mp_elems, new_bindings, matched_elem := \
              unbound_func (bindings, element){
              if bindings.contains("n") & \
                not bindings["n"] == element:
                return Input_Pattern_Creator.fail
              return [Set{}, Dict{"n" : element}, element]
            }(bindings, slot_element):
              if slot_element == matched_elem:
                yield [matched_mp_elems, new_bindings, \
                  matched_elem]
            return Input_Pattern_Creator.fail
          }($$21$$) & $$22$$ := $$21$$ + $$23$$[1] & $$30$$ := \
            unbound_func (bindings){
            slot_element := element.type
            for matched_mp_elems, new_bindings, matched_elem := \
              unbound_func (bindings, element){
              if not Input_Pattern_Creator.TM.type_match("Class", \
                element):
                return Input_Pattern_Creator.fail
              for yield $$24$$ := bindings & \
                $$26$$ := unbound_func (bindings){
                slot_element := element.name
                for matched_mp_elems, new_bindings, matched_elem \
                  := unbound_func (bindings, element){
                  if bindings.contains("cn") & \
                    not bindings["cn"] == element:
                    return Input_Pattern_Creator.fail
                  return [Set{}, Dict{"cn" : element}, element]
                }(bindings, slot_element):
                  if slot_element == matched_elem:
                    yield [matched_mp_elems, new_bindings, \
                      matched_elem]
                return Input_Pattern_Creator.fail
              }($$24$$) & $$25$$ := $$24$$ + $$26$$[1] & \
                $$28$$ := unbound_func (bindings){
                slot_element := element.attrs
                for matched_mp_elems, new_bindings, matched_elem \
                  := unbound_func (bindings, element){
                  if bindings.contains("CA") & \
                    not bindings["CA"] == element:
                    return Input_Pattern_Creator.fail
                  return [Set{}, Dict{"CA" : element}, element]
                }(bindings, slot_element):
                  if slot_element == matched_elem:
                    yield [matched_mp_elems, new_bindings, \
                      matched_elem]
                return Input_Pattern_Creator.fail
              }($$25$$) & $$27$$ := $$25$$ + $$28$$[1] & \
                [Set{element}, \
                Input_Pattern_Creator.Functional.foldl( \
                  Input_Pattern_Creator._adder, \
                Input_Pattern_Creator.Functional.map( \
                Input_Pattern_Creator._element1, \
                [$$26$$, $$28$$])), element]
              return Input_Pattern_Creator.fail
            }(bindings, slot_element):
              if slot_element == matched_elem:
                yield [matched_mp_elems, new_bindings, \
                  matched_elem]
            return Input_Pattern_Creator.fail
          }($$22$$) & $$29$$ := $$22$$ + $$30$$[1] & \
            [Set{element}, \
            Input_Pattern_Creator.Functional.foldl( \
            Input_Pattern_Creator._adder, \
            Input_Pattern_Creator.Functional.map( \
            Input_Pattern_Creator._element1, [$$23$$, $$30$$])), \
            element]
          return Input_Pattern_Creator.fail
        }(bindings, element)
        return Input_Pattern_Creator.fail
      }(bindings, $$31$$elements$$):
        matched_mp_elems := matched_mp_elems + \
          $$34$$new_matched_mp_elems$$
        bindings := bindings + $$35$$new_bindings$$
        return [matched_mp_elems, bindings]
        matched_mp_elems := $$32$$matched_mp_elems_backup$$
        bindings := $$33$$bindings_backup$$
      matched_mp_elems := $$38$$matched_mp_elems_backup$$
      bindings := $$39$$bindings_backup$$
    return Input_Pattern_Creator.fail
  }.apply(objs):
    self._matched_objs.extend(matched_objs)
    if not rtn := unbound_func (matched_objs, bindings){
      concatted_id := Output_Pattern_Creator.concat_id(matched_objs)
      if self._transformed_cache["User_Type_Attr_To_Column"]. \
        contains(concatted_id):
        return self._transformed_cache["User_Type_Attr_To_Column"] \
          [concatted_id]
      tracing_i := self._tracing.len()
      $$43$$prefix$$ := bindings["prefix"]
      $$44$$n$$ := bindings["n"]
      $$45$$cn$$ := bindings["cn"]
      $$46$$CA$$ := bindings["CA"]
      out_elems := []
      out_elems.append(unbound_func (){
        output_objs := []
        for $$47$$ca$$ := $$46$$CA$$.iterate():
          output_objs.append($$49$$self$$.transform( \
            [$$48$$concat_name$$($$43$$prefix$$, $$44$$n$$)], \
            [$$47$$ca$$]))
        return output_objs
      }())
      out_elem := out_elems[0]
      self._transformed_cache["User_Type_Attr_To_Column"][concatted_id] \
        := out_elem
      if not out_elem is Output_Pattern_Creator.null:
        tracing := [Output_Pattern_Creator.List(matched_objs), out_elems]
        tracing := Output_Pattern_Creator.trace_reduce(tracing)
        self._tracing.insert(tracing_i, tracing)
        self._tracing_rule.insert(tracing_i, "User_Type_Attr_To_Column")
      return out_elem
    }(matched_objs, bindings):
      raise MT.Exceptions.Exception(MT.Strings.format("Output pattern" + \
        "of User_Type_Attr_To_Column failed to generate anything for" + \
        " ’%s’.", objs.to_str()))
    return rtn
  else:
    return MT.fail
}
```

```
bound_func Primitive_Type_Attr_To_Column(*objs){
  $$75$$concat_name$$ := concat_name
  if matched_objs, bindings := unbound_func (*args){
    if args.len() > 2:
      return Input_Pattern_Creator.fail
    matched_mp_elems := Set{}
    bindings := Dict{}
    if 0 < args.len():
      $$66$$elements$$ := args[0]
    else:
      $$66$$elements$$ := self._root_set
    $$67$$matched_mp_elems_backup$$ := matched_mp_elems
    $$68$$bindings_backup$$ := bindings
    for $$69$$new_matched_mp_elems$$, $$70$$new_bindings$$, \
      $$71$$matched_elem$$ := unbound_func (bindings, elements){
      for element := elements.iterate() & yield \
        unbound_func (bindings, element){
        if not Input_Pattern_Creator.TM.type_match("String", element):
          return Input_Pattern_Creator.fail
        for yield $$51$$ := unbound_func (bindings, element){
          if bindings.contains("prefix") & \
            not bindings["prefix"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"prefix" : element}, element]
        }(bindings, element) & $$50$$ := bindings + $$51$$[1] & \
          [Set{element}, \
          Input_Pattern_Creator.Functional.foldl( \
          Input_Pattern_Creator._adder, \
          Input_Pattern_Creator.Functional.map( \
          Input_Pattern_Creator._element1, [$$51$$])), element]
        return Input_Pattern_Creator.fail
      }(bindings, element)
      return Input_Pattern_Creator.fail
    }(bindings, $$66$$elements$$):
      matched_mp_elems := matched_mp_elems + $$69$$new_matched_mp_elems$$
      bindings := bindings + $$70$$new_bindings$$
      if 1 < args.len():
        $$60$$elements$$ := args[1]
      else:
        $$60$$elements$$ := self._root_set
      $$61$$matched_mp_elems_backup$$ := matched_mp_elems
      $$62$$bindings_backup$$ := bindings
      for $$63$$new_matched_mp_elems$$, $$64$$new_bindings$$, \
        $$65$$matched_elem$$ := unbound_func (bindings, elements){
        for element := elements.iterate() & yield \
          unbound_func (bindings, element){
          if not Input_Pattern_Creator.TM.type_match("Attribute", \
            element):
            return Input_Pattern_Creator.fail
          for yield $$52$$ := bindings & $$54$$ := \
            unbound_func (bindings){
            slot_element := element.name
            for matched_mp_elems, new_bindings, matched_elem \
              := unbound_func (bindings, element){
              if bindings.contains("n") & \
                not bindings["n"] == element:
                return Input_Pattern_Creator.fail
              return [Set{}, Dict{"n" : element}, element]
            }(bindings, slot_element):
              if slot_element == matched_elem:
                yield [matched_mp_elems, new_bindings, \
                  matched_elem]
            return Input_Pattern_Creator.fail
          }($$52$$) & $$53$$ := $$52$$ + $$54$$[1] & $$59$$ := \
            unbound_func (bindings){
            slot_element := element.type
            for matched_mp_elems, new_bindings, matched_elem := \
              unbound_func (bindings, element){
              if not Input_Pattern_Creator.TM.type_match( \
                "PrimitiveDataType", element):
                return Input_Pattern_Creator.fail
            for yield $$55$$ := bindings & $$57$$ := \
              unbound_func (bindings){
              slot_element := element.name
              for matched_mp_elems, new_bindings, \
                matched_elem := unbound_func (bindings, \
                  element){
                if bindings.contains("pn") & \
                  not bindings["pn"] == element:
                  return Input_Pattern_Creator.fail
                return [Set{}, Dict{"pn" : element}, \
                  element]
              }(bindings, slot_element):
                if slot_element == matched_elem:
                  yield [matched_mp_elems, new_bindings, \
                    matched_elem]
              return Input_Pattern_Creator.fail
            }($$55$$) & $$56$$ := $$55$$ + $$57$$[1] & \
              [Set{element}, \
              Input_Pattern_Creator.Functional.foldl( \
              Input_Pattern_Creator._adder, \
              Input_Pattern_Creator.Functional.map( \
              Input_Pattern_Creator._element1, [$$57$$])), \
              element]
            return Input_Pattern_Creator.fail
          }(bindings, slot_element):
            if slot_element == matched_elem:
              yield [matched_mp_elems, new_bindings, \
                matched_elem]
          return Input_Pattern_Creator.fail
        }($$53$$) & $$58$$ := $$53$$ + $$59$$[1] & \
          [Set{element}, \
          Input_Pattern_Creator.Functional.foldl( \
          Input_Pattern_Creator._adder, \
          Input_Pattern_Creator.Functional.map( \
          Input_Pattern_Creator._element1, [$$54$$, $$59$$])), \
          element]
        return Input_Pattern_Creator.fail
      }(bindings, element)
      return Input_Pattern_Creator.fail
    }(bindings, $$60$$elements$$):
      matched_mp_elems := matched_mp_elems + \
        $$63$$new_matched_mp_elems$$
      bindings := bindings + $$64$$new_bindings$$
      return [matched_mp_elems, bindings]
      matched_mp_elems := $$61$$matched_mp_elems_backup$$
      bindings := $$62$$bindings_backup$$
    matched_mp_elems := $$67$$matched_mp_elems_backup$$
    bindings := $$68$$bindings_backup$$
  return Input_Pattern_Creator.fail
}.apply(objs):
  self._matched_objs.extend(matched_objs)
  if not rtn := unbound_func (matched_objs, bindings){
    concatted_id := Output_Pattern_Creator.concat_id(matched_objs)
    if self._transformed_cache["Primitive_Type_Attr_To_Column"]. \
      contains(concatted_id):
      return self._transformed_cache["Primitive_Type_Attr_To_Column"] \
        [concatted_id]
    tracing_i := self._tracing.len()
    $$72$$prefix$$ := bindings["prefix"]
    $$73$$n$$ := bindings["n"]
    $$74$$pn$$ := bindings["pn"]
    out_elems := []
    out_elems.append([unbound_func (){
      user_args := Dict{"name" : $$75$$concat_name$$($$72$$prefix$$, \
        $$73$$n$$), "type" : $$74$$pn$$}
      all_args := []
      args_processed := 0
      for attr_name := Output_Pattern_Creator.TM.all_attrs_in_order( \
        Output_Pattern_Creator.TM._CLASSES_REPOSITORY["Column"]). \
        iterate():
        if args_processed == user_args.len():
            break
          if user_args.contains(attr_name):
            all_args.append(user_args[attr_name])
            args_processed += 1
          else:
            all_args.append(Output_Pattern_Creator.null)
        return Output_Pattern_Creator.TM._CLASSES_REPOSITORY["Column"]. \
          new.apply(all_args)
      }()])
      out_elem := out_elems[0]
      self._transformed_cache["Primitive_Type_Attr_To_Column"] \
        [concatted_id] := out_elem
      if not out_elem is Output_Pattern_Creator.null:
        tracing := [Output_Pattern_Creator.List(matched_objs), out_elems]
        tracing := Output_Pattern_Creator.trace_reduce(tracing)
        self._tracing.insert(tracing_i, tracing)
        self._tracing_rule.insert(tracing_i, \
          "Primitive_Type_Attr_To_Column")
      return out_elem
    }(matched_objs, bindings):
      raise MT.Exceptions.Exception(MT.Strings.format("Output pattern of" +
        " Primitive_Type_Attr_To_Column failed to generate anything for" +
        " ’%s’.", objs.to_str()))
    return rtn
  else:
    return MT.fail
}
```

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[Big98] Ted J. Biggerstaff. Pattern matching for program generation: A user manual. Technical Report TR-98-55, Microsoft Research, 1998.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, September 2004. Available from `http://www.xactium.com/` Accessed Sep 22 2004.

[Cor04] James R. Cordy. TXL - a language for programming language tools and applications. In *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*, April 2004.

[DIC03] DSTC, IBM, and CBOP. MOF query / views / transformations first revised submission, August 2003. OMG document `ad/2003-08-03`.

[OMG00] Object Management Group. *Meta Object Facility (MOF) Specification*, 2000. `formal/00-04-03`.

[OQV03] OpenQVT. Response to the MOF 2.0 query / views / transformations RFP, August 2003. OMG document `ad/2003-08-05`.

[QVT03a] QVT-Partners. First revised submission to QVT RFP, August 2003. OMG document `ad/03-08-08`.

[QVT03b] QVT-Partners initial submission to QVT RFP, 2003. OMG document `ad/03-03-27`.

[TC03] Laurence Tratt and Tony Clark. Issues surrounding model consistency and QVT. Technical Report TR-03-08, Department of Computer Science, King's College London, December 2003.

[Tra05] Laurence Tratt. The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London, February 2005.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.