A change propagating model transformation language

Technical report TR-06-07, Department of Computer Science, King's College London

Laurence Tratt
laurie@tratt.net

August 13, 2006

Contents

1. Introduction

This paper builds upon the MT language defined in the previous paper, creating a new unidirectional change propagating model transformation language PMT. Alanen and Porres provide a useful overview of change propagating transformations, which also explains some of the categories of changes that can be propagated [AP04]. Change propagating transformations introduce considerable complexity compared to stateless transformations. It is my belief that no one approach to change propagation is likely to prove sufficient for all purposes. Furthermore due to the lack of focus on this particular area of model transformations, much exploration will be necessary to determine when different approaches are most applicable. The aim of this paper is to outline some of the possibilities for change propagating approaches, and to present a particular unidirectional change propagating solution, PMT. PMT is intended to provide support for use cases similar to that outlined in [Tra05b].

Although several model transformation approaches mention change propagating transformations few actually provide such a mechanism. For the purposes of this paper, only three approaches are potentially of interest: BOTL [BM03], Johann and Egyed's approach [JE04], and XMOF [CS03]. Both BOTL and XMOF are of limited interest, due to their differing aims compared to PMT. Since BOTL restricts itself to bijective transformations, I discount it, since I believe that bijective transformations constitute only a small proportion of useful transformations. XMOF is also of limited interest since it is poorly documented, and aims to provide a solution for bidirectional change propagating model transformations, which introduces an extra set of challenges above and beyond those presented by unidirectional change propagating model transformations. Johann and Egyed's approach is the most interesting of the three, as it tackles unidirectional change propagating model transformations; however it explains only one aspect of its approach in detail, and furthermore is incapable of propagating some important types of changes.

It is an explicit aim of PMT to facilitate change propagation in any type of model transformation. However it is important to note that PMT is not as mature or stable as MT – by its nature PMT is much more of an experiment than MT. Nevertheless I hope that this paper serves as a useful step on the path towards mature change propagating model transformation solutions.

This paper begins with an overview of some of the high-level strategies and design decisions relevant to change propagation. PMT itself is then introduced, and via example it is shown how it allows change propagating transformations to be expressed. I show how PMT relates source and target models, and how it is capable of propagating changes that defeat other approaches. I also detail PMT's support for expressing change propagating transformation specifications. Finally I detail some of the relevant parts of PMT's implementation.

2. Change propagation

Whilst [Tra05b] motivated change propagating model transformations, it gave very little hint as to how such transformations might be realised. The intention of this section is to outline the background of change propagations, and some of the overall design decisions possible when implementing a change propagating model transformation approach. Note that I only consider these design decisions in the context of unidirectional change propagating transformations.

2.1. Change propagation compared to incremental transformation

Incremental transformation (sometimes known as incremental computation) is a well studied field (see [RR93] for an overview of some of the available literature). The most widespread, and one of the simplest, examples of incremental transformation are code compilation systems. For example the UNIX `make` command takes a list of source code files, and compiles only those which have been modified since the last execution of `make`.

Incremental transformation initially appears to be very similar to change propagation. Both approaches provide support for taking a source item and transforming it into an appropriate target item; subsequent changes made to the source item then cause appropriate updates on the target item. However incremental transformation approaches assume that the target item will be unmodified by the user when they update it. Incremental transformation need not therefore concern itself with many of the issues that affect change propagation in the context of this paper, chiefly how to propagate changes non-destructively into the target model. This can be seen clearly in the code compilation system example; any modifications the user may make to the output of the compiler will be lost the next time the code compilation system discovers it needs to recompile the associated source file.

There is thus a fundamental difference between the two approaches, since an incremental transformation approach is able to make assumptions about its environment that conflict with the use case outlined in [Tra05b]. For the purposes of this paper, change propagation is therefore largely treated as a new subject with respect to incremental transformation systems.

2.2. Manual or automatic change propagation

Tratt and Clark outline a framework intended to allow unidirectional stateless transformations to be associated with one or more *delta transformations* which can propagate changes [TC03]. The execution sequence of such transformations is as follows. The unidirectional stateless transformation takes in a source model and produces a target model as normal. Subsequent changes made to the source model are extracted as change deltas to the source model. These deltas are then passed to an appropriate delta transformation which is expected to propagate the change represented by the delta to the target model. In general each different type of change will require a different delta transformation to be created. Note that the framework itself does not impose, or facilitate, a particular change propagation mechanism is left open in this framework. An example of this framework can be seen in the XMF tool which includes a change propagation framework with a dedicated delta transformation language XSync, to accompany a unidirectional stateless model transformation language XMap [CESW04].

The concept of delta transformations is an interesting one in that it provides a means of integrating legacy, or otherwise incapable, transformations into a change propagating transformation. However it has two inherent problems. Firstly there is an inevitable disconnect between the core unidirectional stateless transformation, and the delta transformations, all of which must be created by hand. Secondly there is, in general, no bound on the number of delta transformations needed to cope with change deltas. For this reason I classify this framework as manual change propagation, since the code to perform change propagation must be manually created.

Manual change propagation contrasts with automatic change propagation, where a transformation can propagate changes without additional code needing to be added. Some approaches choose a hybrid approach, being able to automatically propagate some changes whilst requiring manual assistance to propagate others. For example, OptimalJ is able to propagate changes between some of its simple models auto-

matically, but can require assistance when propagating changes between a complex model and its textual representation [OJ04].

2.3. Propagating changes in batch or immediate mode

There are two potential modes of operation for running change propagating transformations: 'batch' and 'immediate' mode. These two modes refer to the number of changes that are propagated in each step.

Batch change propagation takes a number of changes from the source model and propagates them to the target model only when explicitly requested to do so by the user. The advantage of batch change propagation is that the user is in complete control of when changes are propagated. Batch change propagation can be considered to be similar to code compilation — users typically make multiple edits to a source code file before choosing to compile it. Since change propagation may be a relatively slow activity, it is beneficial to the user if they can schedule change propagation at a time convenient to them. On the other hand, the user may consider it inconvenient to have to manually force changes to be propagated.

The concept of immediate change propagating transformations is defined in [CS03]. An immediate change propagating transformation propagates changes to the target model as soon as the source model is changed. Unlike a batch mode change propagating transformation, which implicitly propagates multiple changes when run, an immediate mode change propagating transformation propagates small changes, which can be viewed as being semi-atomic. The advantage of immediate mode propagation is that the source and target models involved in the transformation are always synchronised with each other. However there are a number of potential disadvantages to immediate change propagating transformations.

From the users point of view, immediate change propagation may introduce a lag every time the user makes a change to the source model, whilst the system propagates the appropriate changes to the target model. During this lag, the system can choose to either lock the source model, thus preventing the user making changes to it, or to place changes to the source model into an ordered queue. In the former case, the user is likely to become highly frustrated; in the latter case, the advantage of synchronised source and target models is lost, albeit temporarily. Furthermore, the process of changing a model frequently involves passing through one or more intermediate stages. Each intermediate stage may see elements being temporarily deleted, renamed and so on. If the changes from these intermediate stages are propagated, it is possible that incorrect, and irreversible, changes may be made to the target model. Consider a tool which allows a user to 'cut' a model element to a clipboard, who then intends to paste the element to another part of the model later. If such a change is propagated immediately, it will lead to the deletion of target elements. Such elements may contain manual changes or additions in the target model; when the element is deleted, the manual changes will be lost and will not be replaced when the source element is 'pasted' back into the model. Since only the user can know the intended end goal of their sequence of actions, immediate change propagating transformations pose an extra set of challenges for such scenarios.

2.4. Relating source and target elements by key, trace, or identifier

One of the chief challenges when propagating changes is to find a mechanism for relating, or distinguishing, the specific target elements created by a given rule relative to specific source elements. The distinguishing of elements is vital to ensure that target elements are modified, created or deleted correctly during change propagation. This problem is largely irrelevant during the initial run of a change propagating transformation, but is vital when subsequently propagating changes; this problem was outlined by example in [Tra05b].

Johann and Egyed present a basic, high-level overview of this subject, describing the distinguishing of elements by key and by identifier [JE04]. For the purposes of this paper I identify three chief ways of relating or distinguishing which target elements are related to specific source elements: by key, by trace, and by identifier. I now outline these three possibilities in more detail.

2.4.1. Distinguishing target elements by key

A simple mechanism for distinguishing elements is to do so on their key i.e. a collection of attributes which, collectively, uniquely identify any given element. Using this mechanism for change propagation is advocated by the DSTC QVT approach [DIC03]. By requiring elements to be defined in keys, this mechanism implicitly adds an extra burden on the user since all elements in a model must be augmented with a key definition. Although this is often trivial, it is an extra burden, and can be difficult when elements have no natural key.

The essential idea of propagating by key is that when changes from an element need to be propagated, the source element is transformed (possibly to a temporary location), and the key of the target element is extracted. This then allows the changed parts of the target element to be merged with an existing target element with the same key. However this means that modifying the values of attributes involved in a key confuses the propagation algorithm. Consider the transformation from and to a simple modelling language where the key of a `Class` is its `name` attribute. If a class named `x` is transformed to a class also named `x`, then many changes made to the source model (e.g. adding attributes) can be trivially propagated to the relevant target element by transforming the source models key and finding the target element with the appropriate key. However if the source element is renamed to `y` then the key relationship between the source element `y` and target element `x` is broken; the change propagation algorithm will assume that the relevant target element has been deleted, and will recreate it from scratch.

Although not mentioned in the DSTC QVT approach, one technique which may potentially improve the coverage of this technique is to use the previous generation of a source element to calculate the key of the appropriate target element. This allows changes to be propagated successfully even when source elements have had the values of attributes involved in their key altered. However it is unable to cope when manual changes are made to a target elements' key.

In the general case, propagation by key is insufficient. However it may be combined with other propagation techniques to increase coverage.

2.4.2. Relating target elements via tracing information

Using the tracing information created by a transformation to relate source and target elements seems a good candidate, particularly as the information already exists. However, as shown in MT, there are various different tracing information creation mechanisms. The success of a change propagation algorithm then depends on factors such as the coverage and granularity of the recorded tracing information. For example, while the default tracing information generated by MT records which target elements were created by a rule from specific source elements, it does not generate enough information to know which part of the rule created which target element. Such information may be vital for an accurate change propagation algorithm.

There is thus a potential tension between the different uses of tracing information. The type of tracing information desirable for change propagation may be very different from that required by a user to understand transformations on their model. However, assuming that it is suitably detailed, tracing information is sufficient as the sole means of distinguish elements for change propagation.

2.4.3. Distinguishing target elements by identifier

A technique that can ultimately be seen as a slight variation on distinguishing target elements by tracing information was detailed by this author in [Tra05a], and independently by Johann and Egyed in [JE04]. When a target element is created it is given an identifier which contains, at a minimum, the concatenated identifiers of all the source elements which led to the creation of the target element. Henceforth I refer to this as the *target element identifier*. Note that the target element identifier may be in addition to an elements standard identifier, and that conceptually there is no requirement that this new identifier be a single field.

Conceptually this technique does not add any additional power over using tracing information to distinguish elements; it is an alternative way of storing tracing information. Indeed, a simple concatenation of the source elements identifiers means that the target element identifier is merely an alternative way of storing information that can in theory be directly derived from suitably fine-grained tracing information. However extra information can be easily stored in the target element identifier, if required, to allow a transformation to encode information which may not be present in tracing information. This then allows tracing information to be used for other purposes. Furthermore this then means that tracing information need neither have complete coverage, nor be fine-grained; as such, tracing information can be recorded in a fashion which gives it the greatest utility to the user.

2.5. Correctness checking and conflict resolution

Some changes made to a source model may not be able to be propagated successfully to the target model. For example, when propagating an element newly added to the source model, a conflict may arise with an element already present in the target model. There are three main strategies that can be taken in such cases:

1. Propagate all changes regardless of correctness conditions, accepting that the resulting target model may not match expectations, and may even be ill-formed.

2. Check for the correctness of changes before propagating them; refuse to propagate changes which will violate correctness conditions.

3. Propagate all changes which do not violate correctness conditions; note those which violate such conditions and request manual intervention from the user.

Whilst the first strategy requires little extra support, in the cases of the second and third strategies change propagating model transformation approaches have to decide upon the form of correctness checking, its completeness, and its ability to be controlled by users.

3. PMT

PMT's implementation began as a fork of MT, and can be considered initially to be a superset of MT. Most valid MT transformations can be moved into PMT without syntactic change — when used as a stateless model transformation language, PMT performs largely as MT. When compared to the design decisions detailed in section 2, PMT can be said to be a fully automatic, batch change propagation approach, which distinguishes target elements by their identifiers, and which has user controllable correctness checking built in. The details of this broad overview will be filled in as this paper progresses.

Despite many similarities, the sequence of running a PMT transformation is fundamentally different from MT. An MT transformation is initialized with one or more source elements which are immediately

transformed into target elements. In contrast, a PMT transformation is initialized with a source model, a (possibly empty) target model, and a (possibly empty) set of tracing information. Unlike MT, source elements are not immediately transformed after initialization, waiting for the transformation to be executed by the user. Since none, parts, or all, of the target model may be present after the initialization of the PMT transformation, the concept of rule execution in PMT is markedly different in MT. In MT, when a rules source clauses match its input, the execution of the rule implies the production of new target elements. In PMT, when a rules source clauses match its input, the execution of the rule implies that the target model is modified to make it conformant with respect to the transformation. Although from a naïve users perspective there is a difference between the initial execution of a PMT transformation – which appears to populate an empty target model – and subsequent executions which propagate changes, from PMT's perspective there is no difference between the initial and subsequent executions.

Put crudely, the difference between MT and PMT is that the former is an imperative model transformation language whilst the latter is declarative. Conceptually, the execution of a PMT rule is fundamentally different from MT. When a PMT rule is executed, it attempts to make the necessary changes to the target model to satisfy the rules declaration. This may require elements being added, altered and deleted from the target model. The way in which the relationship between source and target elements is specified, and the process by which the update of the target model occurs are the two defining aspects of PMT.

3.1. A PMT transformation's stages

The stages of a PMT transformation are as follows:

1. Take a source model, and an empty target model and transform the source model. This stage – if taken in isolation and viewed as a black box – is essentially identical to an MT transformation. After the transformation has executed, the source and target models, together with the tracing information created, are stored in some fashion.

2. The user may make arbitrary changes to both the source and target models, independent from one another.

3. The user then requests that the changes they have made to the source model are propagated non-destructively to the target model. The transformation is reinitialized with the updated source and target models, and the tracing information from the previous execution. The execution of the transformation then propagates changes from the source model to the target model. After the transformation has executed, the source and target models, together with the new tracing information created are once again stored.

   At this point, the sequence moves back to stage 2.

3.2. Example

This subsection presents a simple example of change propagation, which is based on the change propagation example of [Tra05b]. That example showed the conceptual problems of a change propagating transformation from the ML2 to the ML1 modelling language. The metamodels of the ML1 and ML2 modelling languages are shown in figures 1 and 2 respectively.

The transformation itself is as follows:

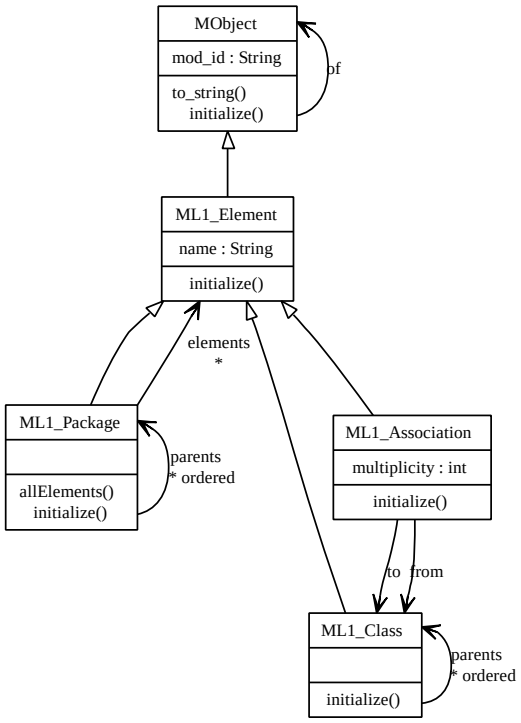


Figure 1: The ML1 modelling language.

```
$<PMT.mt>:
  transformation ML2_to_ML1

  rule Package_To_Package:
    srcp:
      (ML2_Package)[name == <n>, elements == <elements>]

    tgtp:
      (ML1_Package)[name := n, elements :>= tgt_elements]

    tgt_where:
      tgt_elements := Set{}
      for x := elements.iterate():
        tgt_element := self.transform([x])
        if tgt_element.conforms_to(List):
          tgt_elements.extend(Set(tgt_element))
        else:
          tgt_elements.add(tgt_element)

  rule Class_To_Class:
    srcp:
      (ML2_Class)[name == <n>]

    tgtp:
      (ML1_Class)[name := n]

  rule Association_To_Association:
    srcp:
      (ML2_Association)[name == <n>, end1 == <end1>, end2 == <end2>, \
        end1_directed == 0, end2_directed == 0, \
        end1_multiplicity == <end1_multiplicity>, \
        end2_multiplicity == <end2_multiplicity>, end1_name == <end1_name>, \
        end2_name == <end2_name>]
```

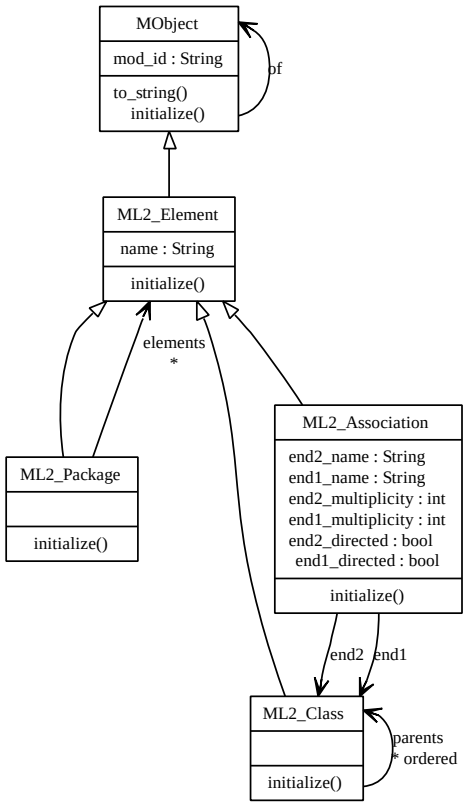


Figure 2: The ML2 modelling language.

```
34
35      tgtp:
36        (ML1_Association)[name := end2_name, from := tgt_end1, to := tgt_end2, \
37          multiplicity := end2_multiplicity]
38        (ML1_Association)[name := end1_name, from := tgt_end2, to := tgt_end1, \
39          multiplicity := end1_multiplicity]
40
41      tgt_where:
42        tgt_end1 := self.transform([end1])
43        tgt_end2 := self.transform([end2])
```

This is an intentionally simple transformation which, in the interests of brevity, ignores parent packages and only handles associations which are navigable at both ends. Since converting ML2 classes and packages to ML1 classes and packages is exceedingly trivial, the `Package_To_Package` and `Class_To_Class` rules are simple (lines 12 - 18 are a largely inconsequential implementation detail that essentially normalizes the return value from other transformation rules). The `Association_To_Association` rule is slightly more complex, although it only deals with associations which are navigable at both ends; each such ML2 bidirectional association is transformed into two ML1 directed associations.

The initial source model I use for this transformation is shown in figure 3. The resulting visualization of the transformation is shown in figure 4. At this point, there are only two hints that we are dealing with a PMT, and not an MT, transformation definition and execution: the `:>=` operator in line 9 is invalid in MT; identifiers in the target model have a noticeably different format to those in MT transformations.

Let us now assume that the user has modified the target model as in figure 5, adding in a directed association from `Employee` to `Manager` denoting an employee's secondary manager. Let us then assume that the user returns to the original source model and updates it as in figure 6, adding in a `DepartmentHead`

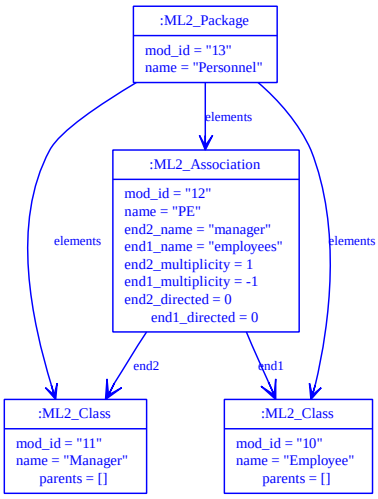


Figure 3: Initial source model for the ML2 to ML1 transformation.

class and an associated transformation. If the ML2 to ML1 transformation was an MT transformation, the user would now have two choices. If they were to rerun such a transformation, the original target model would be overwritten and the `secondary_manager` association would not exist in the new target model. Alternatively the user could choose to manually port the changes from the source model to the target model. In the former scenario, changes to one or the other model are lost; in the latter, differences must be manually propagated between models.

It is at this point – corresponding to stage 3 as described in section 3.1 – in the transformation execution cycle that PMT fundamentally distinguishes itself from MT, by automatically propagating the changes made to the source model in figure 6 into the updated target model. The visualization of the target model after change propagation can be seen in figure 7. As this figure shows, not only have the changes to the source model been propagated into the target model, but the manual changes made to the target model by the user have been preserved. It is important to note that the changes made to the source and target models by the user in this example are entirely arbitrary.

The basics of PMT's change propagation approach are very simple. Both model element patterns and model element expressions play a key part in the process of propagation. PMT uses model element patterns as the primary means of calculating target element identifiers (see section 2.4). When a rule is executed, and its source clauses match successfully, a target element identifier is created, based on unioning the identifiers of the source elements matched by model element expressions. Target element expressions in the target clauses use the target element identifier created by the source clauses. When a model element expression is executed, it looks in the TM object repository to see if an element with the same identifier as the target element identifier already exists. If no such element exists, a new model element with that identifier is created and populated accordingly. If such an element exists, it is taken from the object repository and its contents are adjusted as necessary to satisfy the transformation. Sections 3.3 and 3.4 explain the creation of identifiers and altering of elements in more depth.

3.3. Creating target element identifiers

The construction of target element identifiers is a vital part of PMT's change propagation approach. Target element identifiers should ideally satisfy two criteria: that they are unique with respect to particular source

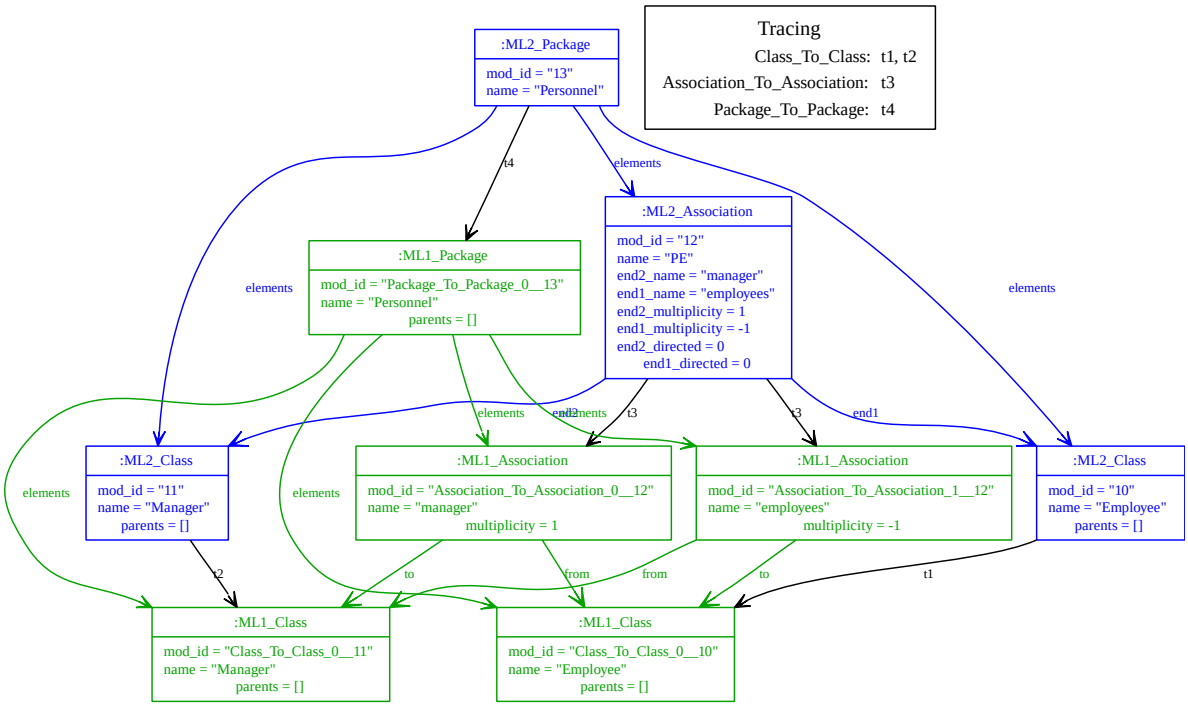


Figure 4: Visualization of the initial execution of the ML2 to ML1 transformation.

elements and a particular rule execution; that they can be created deterministically across multiple transformation executions. The need for the former criteria is self evident, the latter perhaps less so. However PMT's approach relies on the fact that the construction of target element identifiers can be replicated over multiple transformation executions. Since satisfying either, or both, of these two criteria is non-trivial, I consider it highly desirable that target element identifiers can be automatically created and used without burdening the user unnecessarily. In this subsection I outline in detail how PMT automatically creates target element identifiers; this process is somewhat more involved than its description in previous sections has suggested.

The way in which target element identifiers are created and stored makes use of two internal TM and PMT features. Firstly the identifier of a TM model element is a string. Unioning identifiers thus becomes a case of simple string concatenation which, whilst not an entirely robust technique, is adequate for the

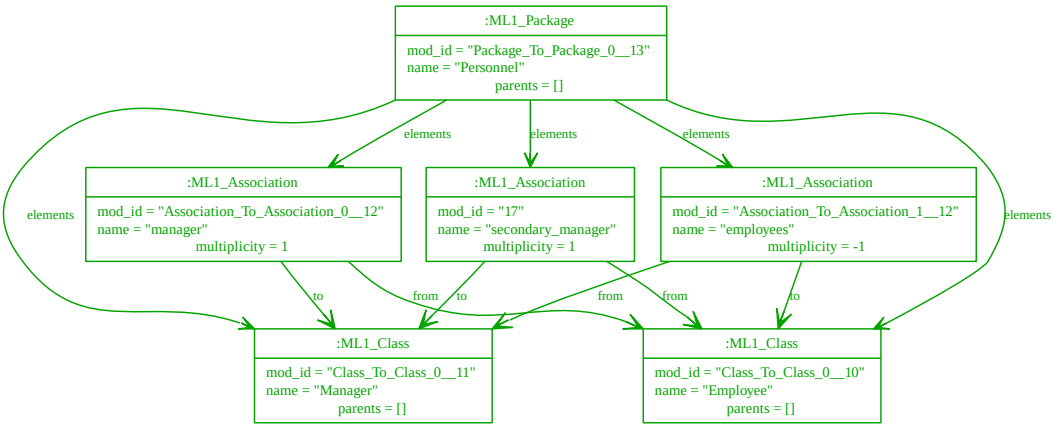


Figure 5: The updated target model.

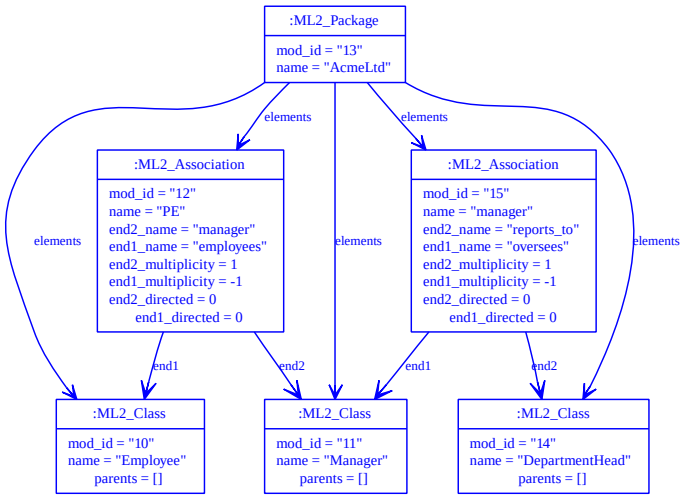


Figure 6: The updated source model.

purposes of this paper. Although TM supplies a default identifier, a user supplied identifier – such as a PMT target element identifier – can be specified when elements are created. Secondly, PMT uses the concept of model elements matched by model element patterns – exactly as used by the tracing information creation mechanism – to determine which source elements will have their identifiers unioned. Thus creating target element identifiers requires no new underlying machinery in the implementation.

3.3.1. Creating unique target element identifiers

Concatenating the identifiers of source elements is not sufficient on its own to generate a unique target element identifier, since the same source elements may be used in more than one rule execution. PMT thus also integrates the name of the rule being executed into the target identifier to ensure that target element identifiers are unique. However this then raises the possibility that executing the same rule with the same source elements may lead to conflicting target identifiers being generated. To avoid this possibility, PMT rules keep a cache of source elements they have already transformed; if a rule matches against the same source elements as it did in a previous execution, then the target elements produced in that previous execution are returned. It should be noted that this is different from MT, which does not need to enforce such a constraint during its execution. This may potentially lead to differences in the execution of seemingly identical MT and PMT transformations.

The rules given thus far generate a single unique target element identifiers. This is sufficient when a rules target clauses contain a single model element expression which executes only once. If a rule has multiple model element expressions in its target clauses, or if a model element expression can execute more than once in a single execution of a rule (e.g. when a model element expression is suffixed with `for`, as in MT), then a single target element identifier would result in multiple target elements being created with the same identifier. For example, the `Association_To_Association` rule in section 3.2 has two model element expressions in its `tgtp` clause. In such cases it is vital that each model element expression is passed a unique target element identifier. In order to ensure that this is the case, each rule execution keeps a counter of how many times model element expressions have been executed during the rules execution. This counter is incorporated into the target element identifier of model element expressions, thus ensuring the uniqueness of the identifiers even when a rule executes more than one model element expression.

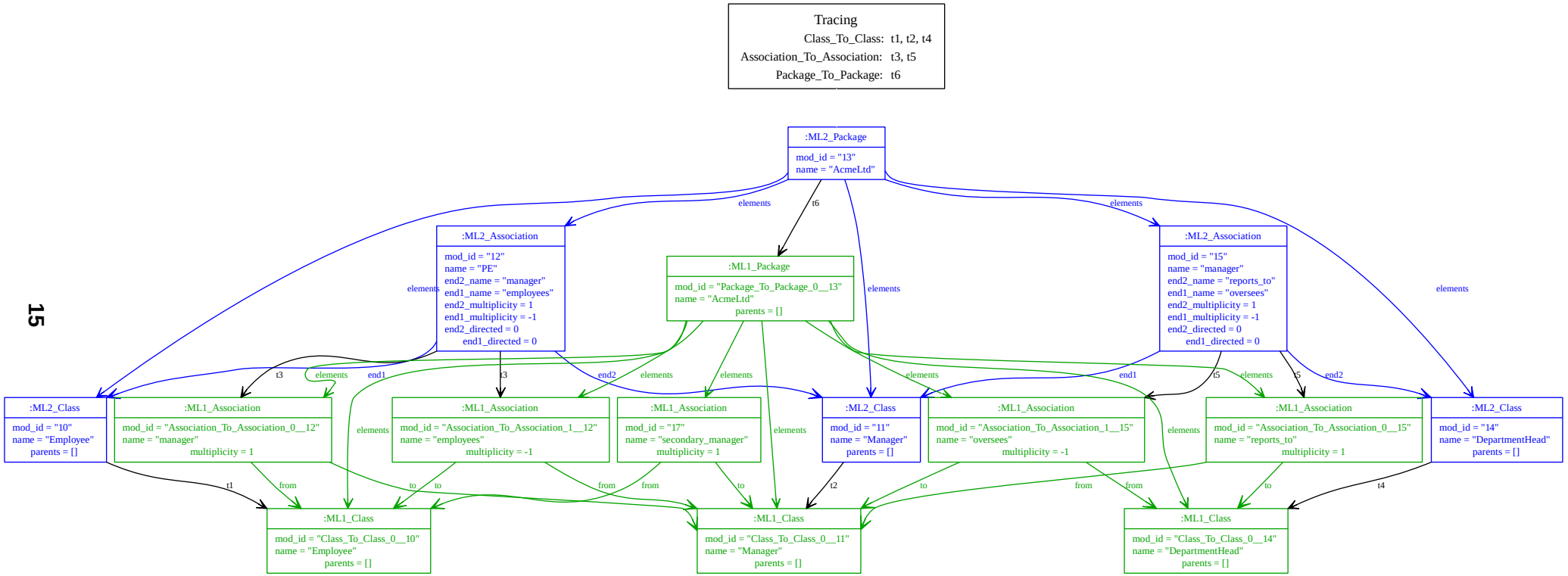


Figure 7: Visualization of the ML2 to ML1 transformation after change propagation.

The general form of a target element identifier in PMT is as follows:

```
<rule name>_<model element expression execution #>__<union of source
  identifiers>
```

Using this template, one can interpret the identifiers of target elements in figure 7 with respect to the transformation of section 3.2.

It should be noted that in the current implementation when primitive data types are used in model element expressions, it is possible for PMT to generate non-unique identifiers, since instances of primitive data types do not have a proper element identifier. I consider this to be a relatively trivial implementation detail.

3.3.2. Deterministically creating target element identifiers

It is important for PMT that the target element identifiers it create be deterministic; that is, if a transformation is rerun with exactly the same source elements as before, it should create exactly the same target element identifiers. If target element identifiers are created differently over multiple transformation executions then PMT will not able to identify target elements correctly. Although the scheme outlined previously has proved reasonably successful in practise, using the model element expression execution counter leads to a subtle, but potentially significant, flaw.

Non-ordered datatypes such as sets can cause the model element expression execution to become de-synchronised over multiple transformation executions due to their inherent non-determinism. Similarly, ordered data types such as lists can have elements inserted in them in-between transformation executions; if elements are inserted at any point other than the end of the ordered datatype, then the counter can again become de-synchronised.

A possible solution to this problem is as follows. Each model element expression in the target clauses is statically assigned a number, starting from 1, and incremented with each model element expression encountered during compilation. For model element expressions that can only be executed once, this is sufficient to ensure uniqueness and determinism of the resultant target element identifiers. For model element expressions which can be executed more than once, it is then necessary to add something further to the target element identifier to ensure uniqueness. For example, one could determine which source elements (which, in general, one would expect to be a strict subset of the overall source elements matched by a rule) led to the creation of that particular model element, and make their identifiers part of the target element identifier; note that in this scheme it would be common for source element identifiers to appear more than once in a target element identifier. In some cases PMT may be able to automatically determine which source elements are involved in the creation of specific target elements, but in general this is not possible; the user will therefore need a way to inform PMT of the required information. Note that whilst this solution is largely immune to non-determinism problems, it still has some conceptual problems e.g. when dealing with ordered lists which contain duplicated elements.

While solutions such as the one outlined may provide a more robust approach to creating target element identifiers, I believe that further research will be needed to find the best solution. For the purposes of this paper, PMT's current solution, whilst not robust, is adequate for exploring change propagation.

3.4. Making target elements conformant

When a model element expression is executed, it looks in the TM object repository to see if an element with the same identifier as the target element identifier already exists. If no such element exists, PMT executes

largely as MT. However if such an element exists, PMT executes rather differently from MT. The object in question is taken from the object repository and PMT and is altered into a form conformant with the model element expression.

It is important to note the use of language in this subsection. When an element already exists it is not necessarily changed to match the exact values dictated by the model element expression. Instead the element has the minimal number of changes applied to it that make it conformant to the model element expression. The word 'conformant' is important since, in the general case, an infinite number of differing target elements may be conformant to a given execution of a model element expression. This is because the user can make manual changes and additions to the target model which the transformation writer can, if they choose, allow to remain even when changes are propagated.

In order to achieve this, model element expressions in PMT have additional syntax compared to MT. Most importantly a model element expression in PMT comprises zero or more *slot conformances* (which are directly analogous to slot comparisons in model element patterns). In the example shown earlier, one can see the use of two *conformance operators*. PMT's conformance operators are partially inspired by operators found in xMOF. Some conformance operators are as follows:

Operator	Name	Description
`x := y`	*update*	Forcibly sets the value of slot `x` to `y`.
`x :== y`	*update if not equal*	If the value of slot `x` is not equal to `y`, forcibly sets the value of slot `x` to `y`.
`x :>= y`	*update superset*	The value of slot `x` must be a non-strict superset of `y`'s value. Any elements in `y` not present in `x` will be added to `x`. `x` may contain elements not present in `y`.

The update conformance operator forcibly propagates changes from the updated source model to the target model. The update if not equal conformance operator performs the same action, but only after checking that the value of the slot in the target element is not equal to the value generated by its associated model expression. In practise, the two operators are very similar; however, since in some cases distinct objects can compare equal the user may wish to specify precisely whether they wish the slot value and model expression to hold exactly the same value, or merely two values which are equal. The update superset conformance operator is more interesting since it does not imply, or force, the value of the slot in the target element to be directly equal to the value generated by the model expression. Instead, the value of the slot in the target element is altered to make sure it contains all the elements that the model expression says it should have; if it has extra elements then those are left intact. In practise this operator is the chief means of allowing changes to be propagated non-destructively.

One important point that may not be immediately obvious is that transformation writers still need to use careful thought to determine when each should be used. For example, an inexperienced transformation writer may choose to use the update operator in all slot conformances, since this will ensure that all changes made to the source model. However if the slot in question contains a set then the users' manual changes made in the target model will be destroyed. In such cases, one would generally expect the transformation writer to use the updating slot conformance operator. In some cases, however, the transformation writer may deliberately wish to ensure that the target model contains the transformed set elements, and nothing else, in which case the update conformance operator is the correct choice. Knowledge of the appropriate situations for each conformance operator is likely to be gathered only through knowledge of the source and target domains, and experience with the change propagating approach.

Later in this paper I will examine other conformance operators. However the three conformance operators detailed in this section are currently the only ones which forcibly alter target elements (the other conformance operators described in section 5 check, rather than enforce, conformance). The reason for this is that, between them, these operators appear to cover a very large part of the spectrum of change propagation – certainly, they are sufficient for all examples in this paper.

3.4.1. Changes which can not be propagated

There are various types of changes which PMT is incapable of propagating. The most obvious class of such problems relates to when the propagation of a change results in an ill-formed model (i.e. one which does not conform to its meta-model). In such cases, a standard TM exception is thrown, and the user is informed. Whilst this is currently a somewhat crude mechanism, it does prevent incorrect target models being created. The checking conformance operators detailed in section 5 provided an alternative means of detecting, and reporting, changes which can not be propagated.

3.5. Running a PMT transformation

Running a PMT transformation is very different to MT (see [Tra05b]), which is largely a direct result of the underlying conceptual difference between a stateless and a change propagating model transformation approach. An MT transformation is passed a source model which it instantly transforms into a target model, creating tracing information as it executes. Since a PMT transformation may be executed multiple times, and since between executions its data may have been serialized to permanent storage, it operates in a fundamentally different fashion.

When run for the first time, a PMT transformation is initialized with only a source model. After the transformation executes, the user can extract the target model and tracing information created during the transformations execution. There are then two scenarios before change propagation will occur. The first scenario is that, whilst the transformation is still ‘active’, the user modifies the source and target models. Propagating changes then becomes a simple case of re-executing the transformation, which will automatically pick up the changes made to the models. The second scenario is that after execution, the source and target models, along with the tracing information, are serialized to a persistent store. The transformation itself is then destroyed. Subsequent executions of the transformation thus require the transformation to be reinitialized with the possibly updated source and target models, and the tracing information (which must not have have been changed), all of which will have been deserialized from their persistent store. Once suitably reinitialized, the transformation can then be executed to propagate changes. Both these scenarios are likely to occur in the real world. Whilst the former scenario is likely to occur in short-lived tasks, or when efficiency is key, the latter scenario reflects the practicalities of long-term use and development of particular models. PMT transformations are designed to deal sensibly with both scenarios.

The code to run the example of section 3.2 looks as follows:

```
employee := ML2.ML2_Class("Employee")
manager := ML2.ML2_Class("Manager")
employee_manager := ML2.ML2_Association("PE", employee, manager, 0, 0, -1, 1, \
  "employees", "manager")
personnel := ML2.ML2_Package("Personnel", Set{employee, manager, \
  employee_manager})

transformation := ML2_to_ML1(personnel)
transformation.do_transform()
```

The unassuming, but important, difference between this and running an MT transformation is the `do_transform` function on a transformation object. This function can potentially be called multiple times. Each time it is called it will propagate changes from the source model to the target model.

Extracting the target model and tracing information from a PMT transformation is identical to MT. For those instances when models need to be serialized to a persistent store, the TM package defines a `Serializer` module. This is capable of serializing (i.e. saving) and deserializing (i.e. loading) models and tracing information via the `serialize`, `serialize_tracing`, `deserialize`, and `deserialize_tracing` functions. A slightly simplified version of the code which serializes the ML2 to ML1 transformation is as follows:

```
src_file.write(Serializer.serialize(transformation.get_source()))
tgt_file.write(Serializer.serialize(transformation.get_target()))
tracing_file.write(Serializer.serialize_tracing(transformation.get_tracing(), \
  transformation.get_tracing_rules()))
```

Appendix B.2 shows the output from serializing the source and target models, and tracing information after the first execution of the example in section 3.2.

Reinitializing a PMT transformation involves initializing the transformation not only with the updated source and target models, but also with the tracing information generated on the previous transformation run. The tracing information generated by the previous execution does not play a direct part in the transformation; it is used to determine which elements can be safely deleted from the target model (see section 3.6). An entirely fresh set of tracing information is generated on each execution. A simplified version of the code which deserializes the ML2 to ML1 transformation, and propagates changes is as follows:

```
src_model := Serializer.deserialize(src_file.read())
tgt_model := Serializer.deserialize(tgt_file.read())
old_tracing, old_tracing_rules := Serializer.deserialize_tracing( \
  tracing_file.read())

transformation := ML2_to_ML1(personnel)
transformation.set_target(tgt_model)
transformation.set_old_tracing(old_tracing)

transformation.do_transform()
```

Models can be transformed, serialized, altered and have changes propagated into them an arbitrary number of times.

3.6. Removing elements from the target model

An important part of change propagation is to ensure that when elements are removed from the source model, target elements which were created by transforming the source elements in question are removed from the target model. This requirement may at first appear to be solved by examining all target elements at the end of a transformation execution, and removing all target elements which were not created as the direct result of transforming one or more source elements. However this simple solution would also delete any elements manually added to the target model by the user, and as such is clearly not suitable for the use cases PMT is aimed at. The critical problem is therefore to distinguish which seemingly superfluous elements in the target model have been manually added by the user, and which are no longer a part of the transformation.

In order to determine which elements can be safely deleted in the target model, PMT utilises tracing information – both that generated by an execution of the transformation, and that generated by its previous

execution. After changes have been propagated, a PMT transformation examines every element in the target model, checking whether it is referenced in either or both of the current and previous tracing information. Based on this, PMT draws a conclusion about the origins of the element and whether it is a candidate for removal. The four possibilities for an element are as follows:

In previous tracing info.?	In current tracing info.?	Conclusion	Candidate for removal?
√	√	Target element previously manually created by PMT.	×
×	√	Target element newly created by PMT.	×
×	×	Target element previous added to target by user.	×
√	×	Target element previously created by PMT; corresponding source element now deleted.	√

Once every element has been examined, PMT performs a garbage collection style 'mark and sweep' [JL99], using the transformed root set of source elements as the starting point. Any self-contained cycle consisting solely of elements marked as being candidates for removal, is then removed from the target model. The need to identify self-contained cycles of such elements is to prevent the removal of elements cause the target model to become ill-formed. This could occur if elements are removed from the model even though they are referred to by other objects. An example of elements being removed after change propagation can be seen in section 4.2.

3.7. Propagating changes between containers

Propagating changes between containers (e.g. sets and lists) raises two challenges not tackled earlier. The first relates to the removal of elements in containers. The second challenge relates to the synchronising of ordered containers. In this subsection I detail PMT's solutions to these challenges.

3.7.1. Removing elements when propagating changes between containers

When elements are deleted from a container in a source model, and that container is transformed into a container in the target model, PMT needs to be able to work out which elements in the target container should be removed. This is a less than easy task because PMT needs to distinguish elements in the target container which have been manually added by the user, and those that are the result of transforming a now absent source element. In order to make this distinction, PMT uses a technique similar to the general element removal technique of section 3.6.

When the updating superset operator attempts to propagate the changes from a container y to a slot x's value in a target element, it first adds every element of y to x's value if it is not already present therein. It then iterates over x's value, noting any elements in x's value which are not present in y. When it finds such elements, it first checks to see if the element is present in the tracing information of the previous transformation execution. If the element is not present, PMT assumes the additional element in x's value is a manually added element, and ignores it. If the element is present in the previous transformation execution's tracing information, PMT assumes that the element was originally added to the container by PMT, and can now be removed from the container.

Due to a lack of sufficiently fine-grained information, this scheme has one notable problem – if a user manually adds a target element into a container, and the source element that led to the creation of that

particular target element is subsequently deleted, then the element will be erroneously removed from the container upon change propagation. Note that does not imply that the element will necessarily be removed from the model; the element will only be removed – in the mark and sweep phase – if its membership of the container was its only reference within the model.

3.7.2. Propagating change in ordered containers

Propagating changes to ordered containers is considerably more complex than into unordered containers. Not only are elements ordered, but the same element may appear more than once. This means that, for example, it is not acceptable to merely check for the existence of a given element, since it may appear more than once. Similarly, between transformation executions, elements may move their position within a list. When a user is adding, removing, or moving elements within an ordered container, the purpose of each individual change is generally self-evident to them. From the point of view of a system viewing an arbitrary number of such changes, any such intentions are lost.

The update superset conformance operator takes a simple minded approach to the problem. Given a target slot x, and an ordered container y, it will ensure that x's value contains every element of y in the order that those elements are contained within y. However it will tolerate an arbitrary number of extra elements within x. Elements from y are added into x as necessary. Looked at a different way, this mechanism ensures that there is an ordered sublist of x which is exactly equal to y. This scheme is less than ideal, since it can lead to an incorrect duplication of elements in the target container.

4. The execution of a PMT transformation

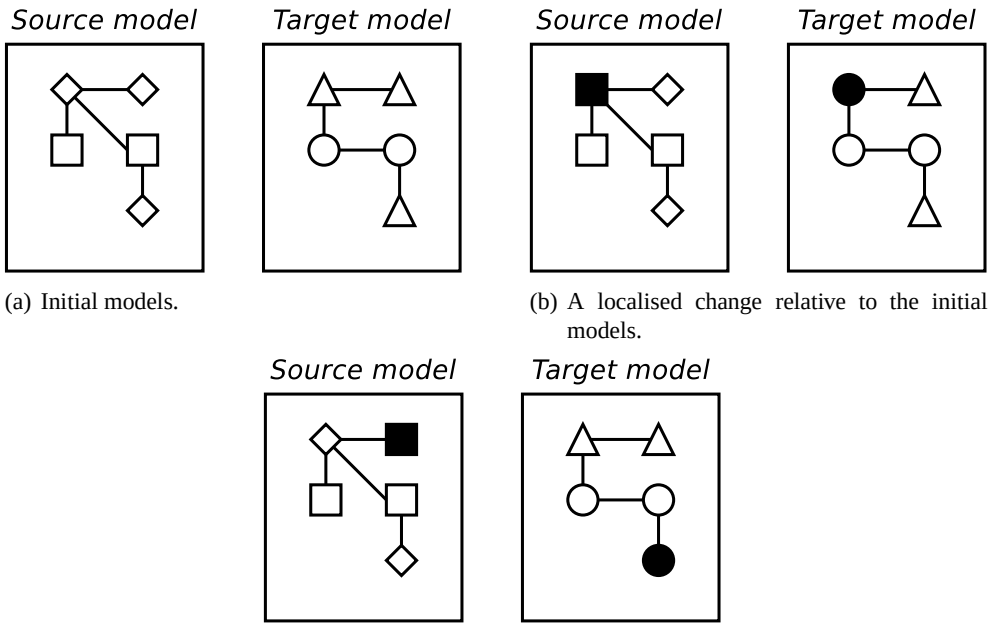
Up until this point I have been deliberately vague on exactly what actually happens when a PMT transformation is executed. The reason for this is that PMT's execution strategy runs contrary to a standard intuition – as exemplified by Johann and Egyed [JE04] – of change propagation in operation. By deferring the explanation of a PMT transformation until this point in the paper, I hope that enough material has been presented to make explanation of this vital point practical.

Intuitively, the concept of change propagation seems simple: given a change in the source model, one simply needs to rerun the few transformation rules which relate to the changed source elements in order to propagate the change to the target model. For many small, localised changes – such as the renaming of a class, as seen in the earlier example in this section – this strategy is adequate. Whilst this intuition is highly appealing, it leads to a solution that can not propagate many types of changes correctly. At best this may lead to a target model that is not synchronised with the source model; at worst, it may cause the target model to become ill-formed.

In this section I first point out the problems with the intuitive change propagation approach, before presenting PMT's approach to transformation execution.

4.1. Propagating localised changes

The change propagating example of section 3.2 saw two main types of changes to the source model: the alteration of the values of elements fields (e.g. changing a packages name), and the addition of elements. The former type of change is intuitively simple to propagate. When the `Personnel` package was renamed to `AcmeLtd` in figure 6, all that is required to propagate the change is to rerun the transformation rule(s)



(a) Initial models.

(b) A localised change relative to the initial models.

(c) A non-localised change relative to the initial models.

Figure 8: The concept of localised changes.

linked to by the tracing from the source element. A quick examination of figure 4 shows that rerunning the `Package_To_Package` rule with the source package in question as input will result in the change being correctly propagated. The latter type of change is slightly more complex, but intuitively somewhat similar. One approach would be to first pass the new source element to the `transform` function; any source elements which have new links to the new element will be transformed using the same approach as for propagating the change in package name.

The fundamental premise behind this intuitive notion is that the propagated changes are what I term *localised*. Note that this term does not directly relate to the locality of alterations in the source model, but instead to the locality of the necessary changes to be propagated to the target model and the relation of those changes to the altered source elements. Figure 8 shows an abstract example of a transformation, and localised and non-localised change propagation. If changes are localised, then changes to elements in the source model can be propagated by rerunning the rules which originally applied to the those elements. This has two implications. Firstly, that changes in the source model will lead to changes in the target model of a similar granularity; in other words, that changes local to a particular part of the source model should lead to similarly local changes in the appropriate part of the target model. The second implication follows from the first: that the source and target models are likely to be mostly, or wholly, isomorphic.

Before I justify these two implications, it is instructive to see why they are implicit in the, rather limited, literature on the subject. For example, Johann and Egyed [JE04] describe a system that is almost wholly targeted at localised changes; despite not being directly model related, Varró and Varró describe a similar system [VV04]. By assuming that changes are localised, both approaches are able to make change propagation highly efficient by only running the rules directly related to a particular change. The ability to highly optimise change propagation in the face of localised changes is a compelling reason to treat such changes as a special case. Unfortunately neither approach is capable of propagating non-localised changes correctly. Johann and Egyed [JE04] describe what they term ‘semantic changes’ as ‘simple changes in the

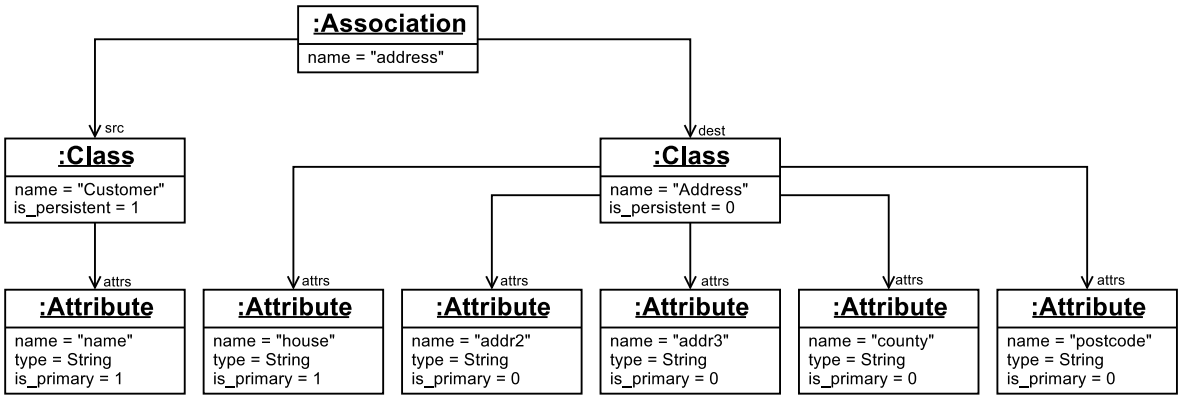


Figure 9: Source model.

source model that cause a variety of ripple effects among multiple/many target elements', but do not present a solution to this problem. I believe the reason for this omission is that many toy transformations, such as the example of section 3.2, are expressed in such a way that only localised changes will ever need to be propagated.

4.1.1. Non-localised changes in practise

Two concrete examples demonstrate the problem of non-localised change. In order to demonstrate this, I return to the advanced variant of the UML modelling language to relational database transformation, as defined in [Tra05b]. I assume that the hypothetical change propagating transformation which would perform this task follows a similar structure to the MT solution for this problem, as defined in [Tra05b].

Consider first the (slightly elided) source model of figure 9, and the corresponding target model in figure 10. Imagine first what would happen were we to change the value of the `is_persistent` slot in `Address` class of figure 9 to `1`. When we execute the transformation to propagate transformations, intuitively we would expect to see the target model contain two tables, and for all the columns prefixed with `address_` to be removed from the `Customer` table. Using a technique similar to that outlined by Johann and Egyed, this intuitive idea may or may not be matched by reality. In the initial transformation execution the `Association_Non_Persistent_Class_To_Columns` would have matched the `Address` class and transformed it. However by marking it as persistent, that rule is no longer able to match (the `Persistent_Association_To_Columns` would however now match), and so change propagation can not occur using the original rule. Johann and Egyed are vague as to what happens when an alteration to the source model means that change propagation can not occur with the original rule which transformed that element. However one can imagine that when such a case is detected the transformation system would look for a different rule which does match the changed source element.

Taking the same source model of figure 9, and the corresponding target model in figure 10 as the basis for the second example, consider the effect of changing the `postcode Attribute`'s `is_primary` key to `1`. Upon change propagation, one would expect to see a new `pkey` link from the `Customer` class to the `address_postcode` column. Assuming, as in the previous example, that alternative rules can be executed when an alteration to a source element invalidates the original rule that transformed it, Johann and Egyed's scheme will not be able to create this link – in fact, the change propagation will not make any changes to the target model at all. This is due to the non-localised nature of the change. Intuitively, although the `postcode Attribute` is changed, the rule which will be rerun (in this case

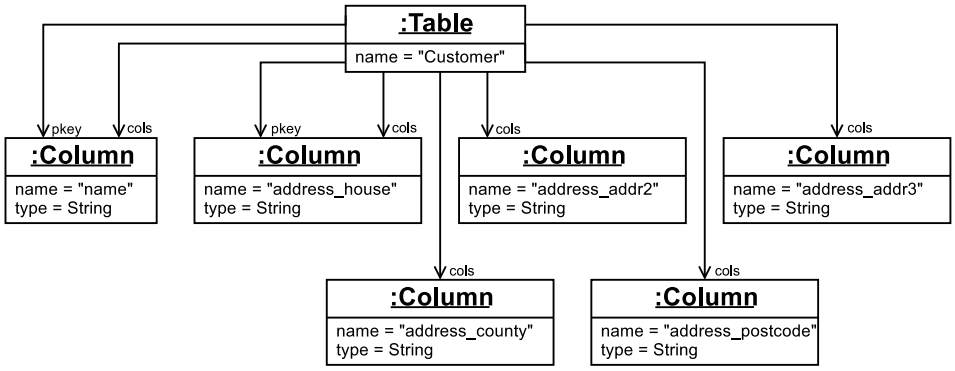


Figure 10: Target model.

`Primary_Primitive_Type_Attribute_To_Columns`) will only transform the `Attribute` itself; any new primary key links it created will be discarded as the transformation will be unaware that the link needs to be considered in an outer context. In other words, although the primary key link will be created, since the transformation rule which transforms classes to tables is not rerun, it will not be incorporated into the transformed table. In general, since the appropriate outer context that needs to be considered may be an arbitrary number of levels away from the element changed, and since the appropriate context can not be determined in advance, rerunning only part of the transformation can never be guaranteed to propagate all changes correctly.

It is left as an exercise to the reader to spot other cases in this example which will similarly foil a change propagation scheme only capable of propagating localised changes. As the examples of this subsection have demonstrated, such schemes have a fundamental weakness when propagating such changes. In the following section, I demonstrate how PMT's more general scheme is capable of propagating such changes correctly.

4.2. PMT's approach

The fundamental challenge with non-localised changes is to determine the particular rules to execute given a particular alteration of the source model. This requires an analysis of all the transformation rules in a system to determine which are relevant to particular changes. In a fully declarative approach such analysis may be possible, although it may be impractical or even impossible depending on the expressive power of the approach. However in a hybrid declarative / imperative approach such as PMT's, analysis of this sort is impossible in the general case – whilst PMT's use of patterns may facilitate analysis in some cases, any use of imperative code (particularly code which calls out to Converge libraries) irreparably muddies the waters. The criteria for PMT's execution approach is thus simple: it must be capable of propagating non-localised changes successfully, and it must be capable of doing so even when it can not analyse the transformation and its rules.

PMT's execution approach thus takes the only solution which can ensure correct operation in all cases: change propagation involves a complete re-execution of the transformation. By executing the transformation from the beginning, PMT implicitly propagates even non-localised changes. The downside to this approach is that rerunning the entire transformation is not efficient. However since PMT is, by design, a batch change propagation approach (see section 2.3), I believe this is considerably less of a problem than it would be for an immediate change propagation approach.

The efficacy of PMT's approach is best seen by example. In order to present a meaningful comparison, I

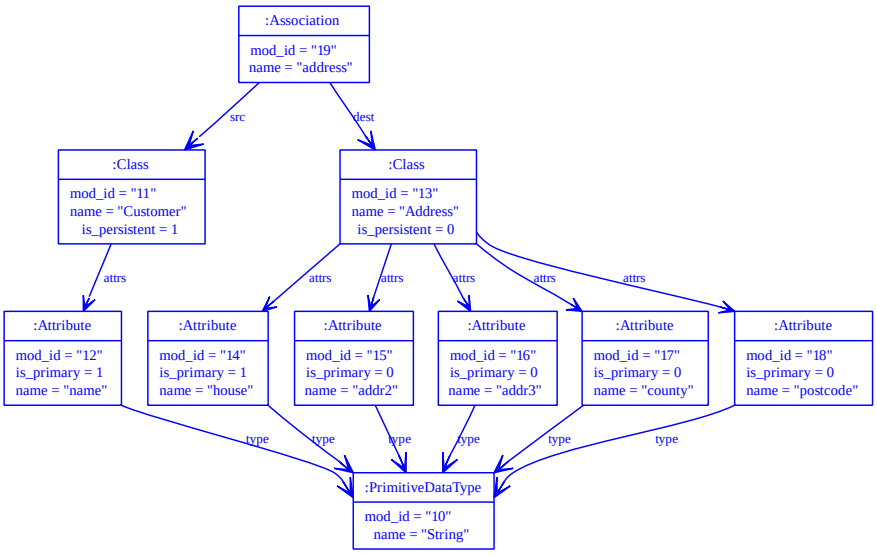


Figure 11: Initial source model.

use exactly the same example as in the previous subsection. In order to have a PMT version of the advanced variant of the UML modelling language to relational database transformation from [Tra05b], one simply needs to substitute `$<PMT.mt>` for `$<MT.mt>` in the transformation code. Although this does not lead to a particularly idiomatic PMT transformation, it saves duplicating the code, and demonstrates how close MT and PMT are in many aspects. Figure 11 shows the initial source model, and figure 12 the target model[1] created by running the `Classes_To_Tables` transformation. Figure 13 shows the updated source model, with the `Address` class marked as being persistent, and the `postcode` attribute marked as being part of a primary key. Figure 14 shows the result of change propagation on the target model.

As this example shows, PMT's change propagation approach ensures that all changes – including non-localised changes – are propagated successfully. I believe the relative inefficiency of this method is thus offset by its ability to propagate non-localised changes correctly. Section 7 discusses potential techniques to increase the efficiency of PMT change propagation in some circumstances.

5. Checking conformance operators

In some situations in a change propagating transformation, the transformation writer may wish to explicitly prevent some types of change propagation from occurring, or ensure that certain relationships between the source and target models always hold. This is potentially very important for PMT's use cases, where the transformation writer may need to constrain the modifications that the user can perform to the target model in order to ensure correct change propagation.

PMT provides support for such use cases by providing *checking* conformance operators (in contrast to the updating conformance operators of section 3.4). By using checking conformance operators, transformation writers are able to write change propagation specifications. Note that any given model element expression may contain updating *and* checking conformance operators; change propagation specifications thus may live directly alongside change propagation implementations.

[1] Note that the occurrence of four '_' characters in target identifiers is the result of an implementation detail regarding the identifier of built-in Converge data types such as strings, and can be safely ignored.

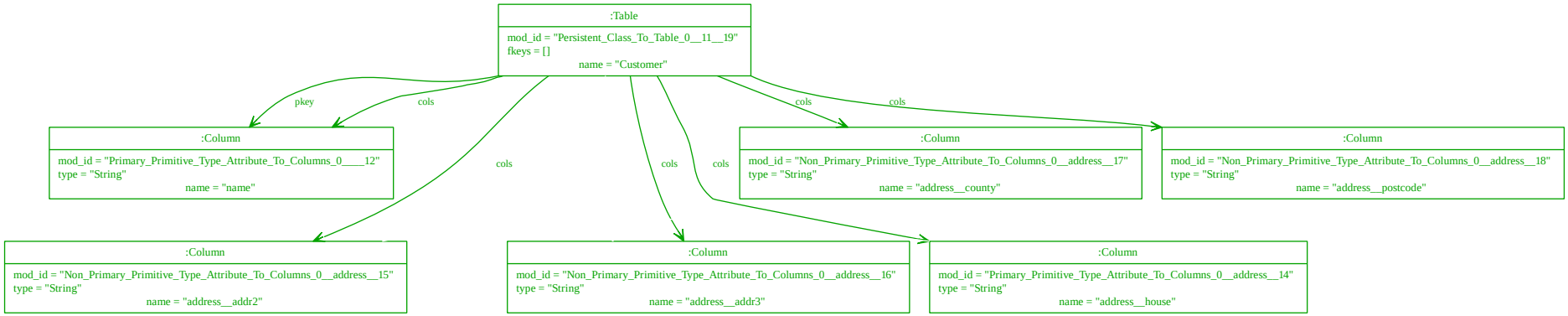


Figure 12: Initial target model.

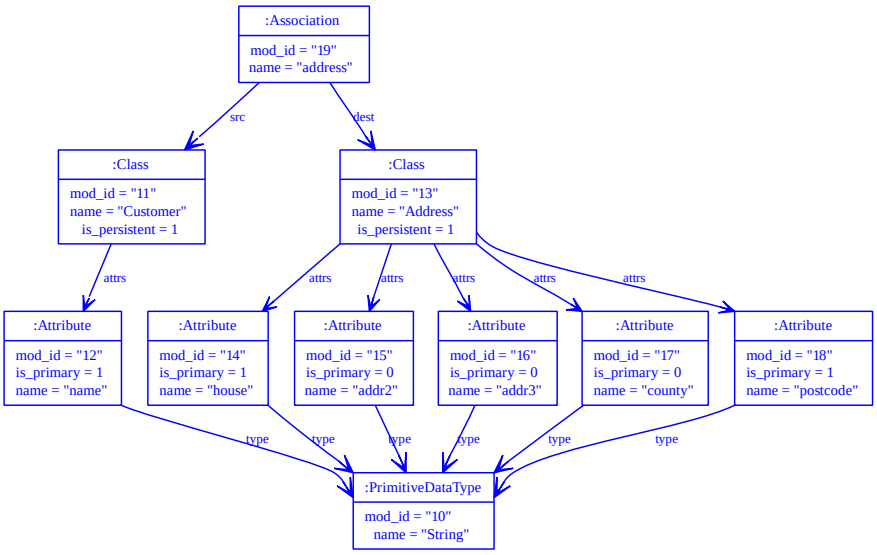


Figure 13: Updated source model before change propagation.

The following checking conformance operators are defined by PMT:

Operator	Name	Description
`x == y`	*equality*	Check that the value of slot `x` is equal to the value of `y`.
`x != y`	*inequality*	Check that the value of slot `x` is not equal to the value of `y`.
`x >= y`	*superset*	Check that the value of slot `x` is a non-strict superset of `y`'s value.
`x <= y`	*subset*	Check that the value of slot `x` is a non-strict subset of `y`'s value.

These operators perform the checks specified in the table, and produce a *conflict report* if the checks fail. A conflict report consists of a number of conflict records. A conflict record pinpoints a specific part of the target model as being non-conformant relative to the rule containing the failing checking conformance operator. Individual conflict records may optionally be able to show what changes would make the target model conformant. The intention of such reports is to report to the user a particular sequence of modifications which, if manually applied to the target model by the user, would make it conformant.

In order to demonstrate checking conformance operators, I once again reuse the example of section 3.2 replacing the `Package_To_Package` rule with the following:

```
rule Package_To_Package:
  srcp:
    (ML2_Package)[name == <n>, elements == <elements>]

  tgtp:
    (ML1_Package)[name == n, elements >= tgt_elements]
```

Essentially this is the same rule as before, but with the updating conformance operators in the `tgtp` clauses' pattern replaced with equivalent checking conformance operators. Similarly I reuse the initial source model of figure 3, which leads to the creation of the same target model as figure 4. I then assume the user alters the target model as per figure 5, and the source model as per figure 6. When propagating changes with the new `Package_To_Package` rule in place, the result of the change propagation is shown in figure 15. Conflicts are clearly shown in red.

The visualization of conflicts in PMT intentionally reuses the visualization techniques from other parts of PMT, with the aim of reducing the learning burden for the user. The 'Conflict report' in figure 15 is

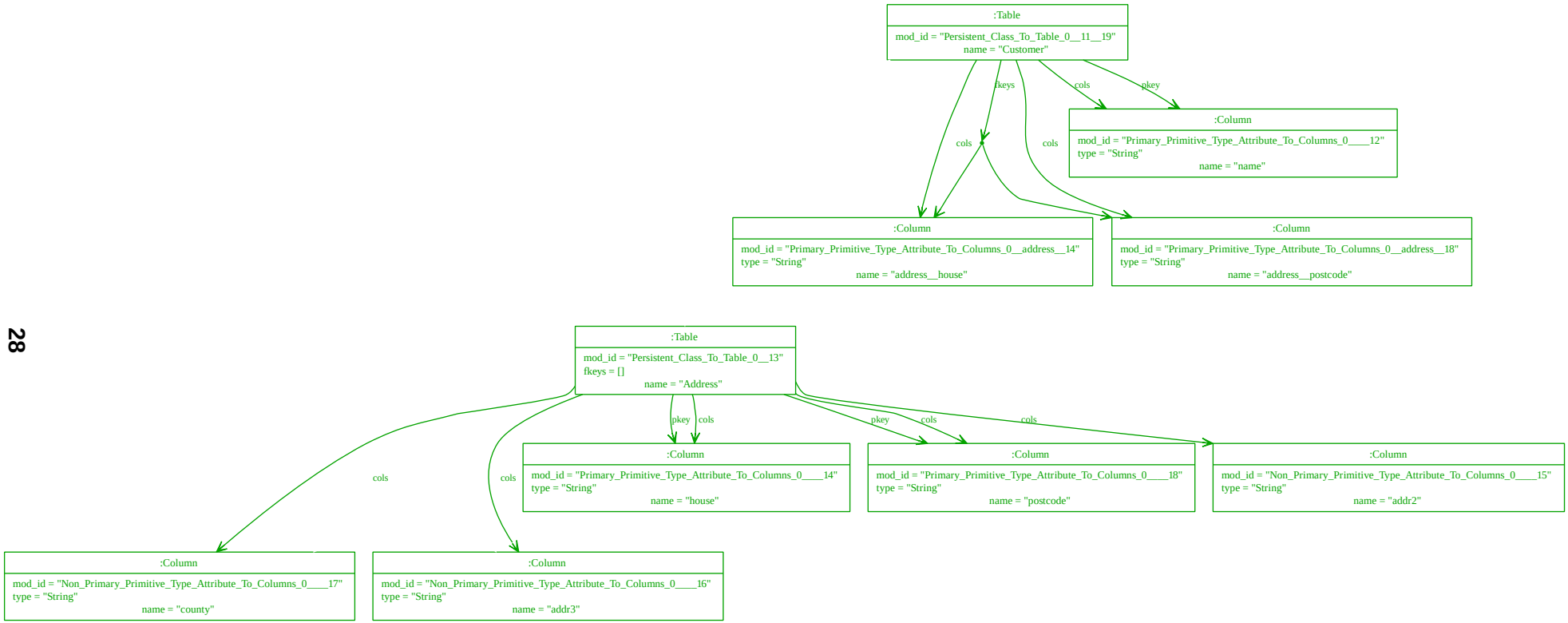


Figure 14: Updated target model after change propagation.

analogous to the ‘Tracing’ report. In a similar fashion to traces, conflicts are named c*n* where *n* is an integer starting from 1. Each separate conflict is generated during a particular execution of a transformation rule. Figure 15 shows two types of conflicts. Conflict ‘c1’ shows that the `name` slot in the `Personnel` package has an incorrect value. Note that the conflict text is surrounded by a rounded box, and the link to the element is a dotted line – these visualizations only occur in conflict reports, and can not be confused with the normal visualization of elements. Conflict ‘c2’ shows elements missing from the `elements` slot of the `Personnel` package. Model elements, and links, in solid (as opposed to broken) red lines show that such elements need to be added to the target model in order to make it conformant. The ‘+’ prefix is a reinforcement of this. Note that the conflict report itself denotes only that the two `ML1_Association` elements, the `ML1_Class` element and the links from the `Personnel` package to those elements, need be added to the target model. However the visualization of the conflict also shows the links between these elements (the `to` and `from` links), since these are implicitly required in order to make the target model well formed. It is important that this information is shown to the user; if it was not, then fixing a conflict report may simply result in another conflict report being generated for a part of the model just added.

Conflict reports create some interesting corner cases. To give a simple example of this, I assume a fresh execution of the `Classes_To_Tables`, once again reusing the initial source and target models of figures 3 and 4 respectively. Removing the `PE` association from the source model and executing the transformation to propagate changes leads to figure 16. The long dashes on the links from the `Personnel` package (combined with the ‘-’ preceding the conflict name on the link) indicate that they should be removed from the target model in order to make it conformant. However one might have expected to see the two `ML1_Association` elements also being drawn in red dashed lines to signify their removal. However, PMT is unable to do this because although the links from the `Personnel` *should* be deleted from the target model, they are not yet deleted. Therefore the two `ML1_Association` elements are reachable via these links and via the garbage collection style algorithm that PMT runs at the end of the transformation (see section 3.6) these two elements are considered to be a valid part of the target model.

Section 6.2 explains the implementation of conflicts in PMT in more detail.

6. Implementation

Unsurprisingly, given its origins, PMT’s implementation is largely similar to MT’s. The majority of PMT’s features are simple changes to MT code using the techniques outlined in [Tra05b], and as such are not documented in detail in this section. Instead I detail two particular parts of PMT’s implementation that are of additional interest over MT’s implementation. PMT’s grammar, which is referenced throughout this section, can be found in appendix A.

6.1. Conformance operators

A simplified version of the `_t_pt_mep_pattern` traversal function, which only contains the code for the `>=` checking conformance operator operating on unordered containers, is given below:

```
1  func _t_pt_mep_pattern(node):
2    // pt_mep_pattern ::= "(" "ID" ")" "[" "ID" pt_mep_pattern_op expr { ","
3    //                    "ID" pt_mep_pattern_op expr }* "]"
4
5    class_ := [| TM._CLASSES_REPOSITORY[$<<CEI.lift(node[2].value)>>] |]
6    conformance_operators := []
```

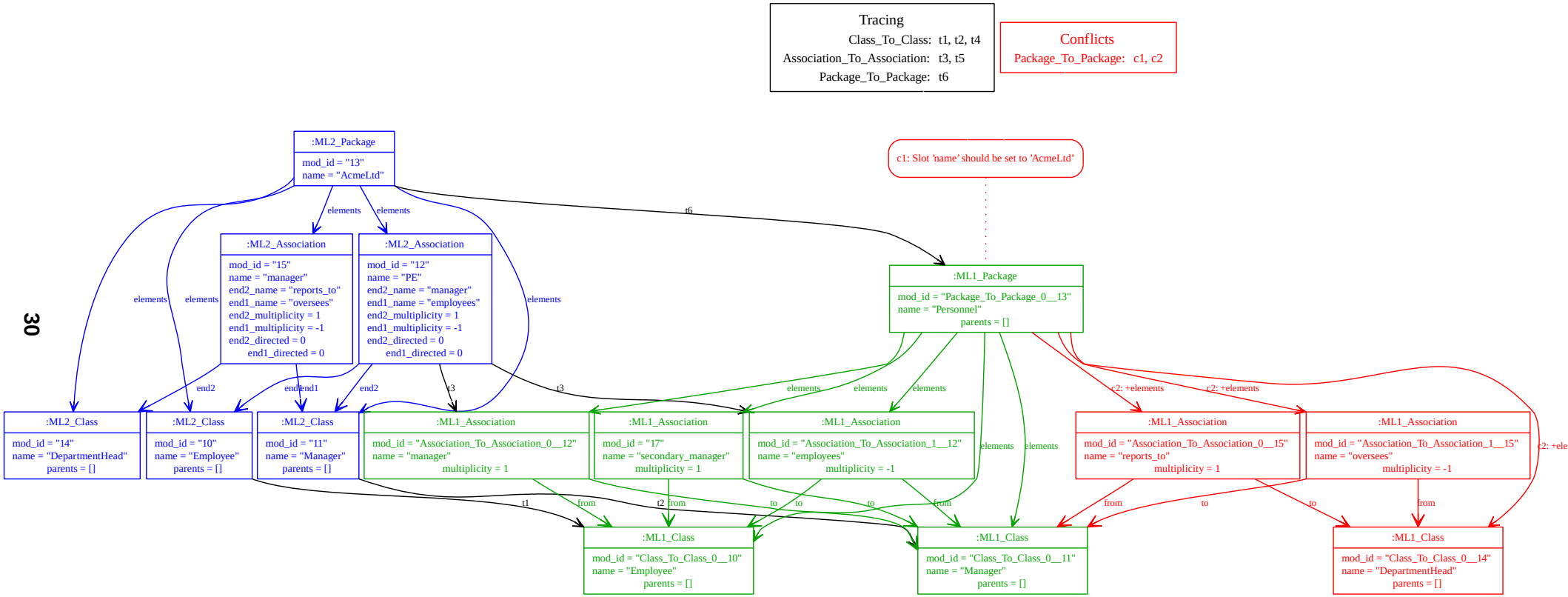


Figure 15: Target model with conflicts.

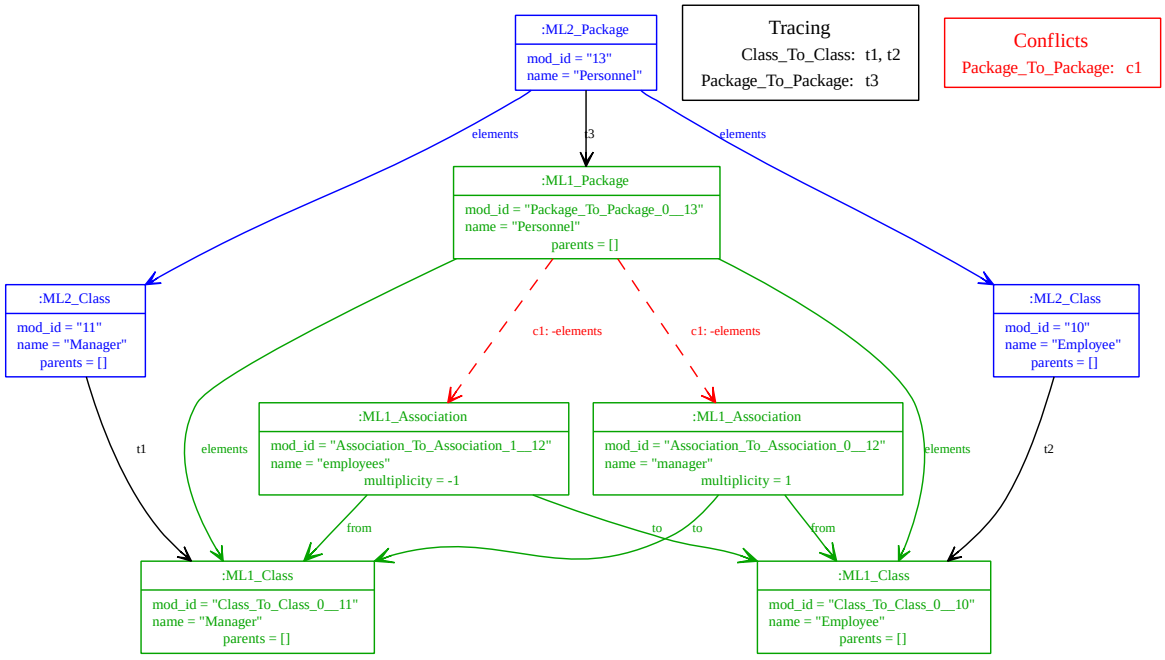


Figure 16: Target model with conflicts after elements are removed from the source model.

```
  i := 5
  while i < node.len() & node[i].type == "ID":
    if node[i + 1][2].type == ">=":
      // pt_mep_pattern_op ::= ":" ">="
      conformance_operators.extend([|
        val := $<<self.preorder(node[i + 2])>>
        if Func_Binding(&obj, Object.fields["get_slot"])("_is_initialized") \
          == 0:
          &obj.$<<CEI.name(node[i].value)>> := val
        elif val.conforms_to(Set):
          should_be_in_the_set := []
          should_not_be_in_the_set := []

          for set_elem := &obj.$<<CEI.name(node[i].value)>>.iterate():
            if not val.contains(set_elem):
              for in_objs, out_objs := &self._old_tracing.iterate():
                if out_objs.contains(set_elem):
                  should_not_be_in_the_set.append(set_elem)
                  break

          for set_elem := val.iterate():
            if not &obj.$<<CEI.name(node[i].value)>>.contains(set_elem):
              should_be_in_the_set.append(set_elem)

          if should_be_in_the_set.len() == 0 & \
            should_not_be_in_the_set.len() == 0:
            pass
          else:
            &self._conflict_objects.append(Conflict.Set_Conflict( \
              $<<CEI.lift(self._rule_name)>>, &matched_objs, &obj, \
              $<<CEI.lift(node[i].value)>>, should_be_in_the_set, \
              should_not_be_in_the_set))
        else:
          raise Type_Exception(Set)
      |])
```

```
42
43    return [|
44      func () {
45
46        new_id := identifier based on rule name union of source elements etc.
47
48        if TM.OBJECTS_REPOSITORY.contains(new_id):
49          obj := TM.OBJECTS_REPOSITORY[new_id]
50        else:
51          obj := $<<class_>>.new_with_id(new_id)
52
53        $<<conformance_operators>>
54
55        return obj
56      }()
57    |]
```

There are two distinct parts to this function. Lines 43 – 55 show the core of the extended model element expressions in PMT. Line 45 calculates the identifier for the model element expression (see section 3.3). Line 47 then checks the TM model element repository to see whether an element with such an identifier already exists. If it does, that element is plucked from the repository (line 48). If it does not, a blank element of the correct type is created (line 50). The element, blank or otherwise, is then handed to the various conformance operators (line 52).

The superset operator is indicative of the the conformance operators in general (sections 3.4 and 5). Firstly the the model element expression is evaluated in line 12. Line 13 then checks to see whether the element has been initialized (meaning that a blank element was created in line 50); if it has not, then the value of the user expression is simply assigned to the appropriate slot and the conformance operator automatically succeeds. If the slot does contain a value, then lines 16 – 37 check the value of the slot for conflicts against the user expression. Lines 19 – 24 check for elements in the slots value that PMT tentatively believes should not be there (see section 3.7.1), while lines 26 – 28 check for elements in the user expression which should be present in the list. If PMT detects that there are elements in the set which should or should not be there, then it generates a conflict report in lines 34 – 37.

6.2. Conflicts

Although conflict reports are generated by PMT, the conflict concept is housed within TM since it needs to understand conflicts in order to be able to visualize them. TM defines a simple model of conflicts which is used to record the required information. Although the model of conflicts is largely an internal detail to PMT and TM, the model presented in this subsection captures the required information in a simple manner; I hope that as other types of conflict reports are needed, it serves as a practical and efficient base for expansion.

TM currently defines three types of conflict records: slot conflicts, list conflicts, and set conflicts. Conflict records conform to the model of figure 17. As this shows, all conflict records share certain things in common. All conflicts are generated from a particular rule (captured by the `rule_name` slot), are the result of transforming one or more source elements (the `src_objs` association), and are specific to a particular `slot_name` within a give target element (the `tgt_obj` association).

Slot conflicts show when a slot with a primitive type (e.g. strings or ints) has an incorrect value. In such a case, the `conflict_obj` records the value the slot should have. List and set conflicts can be considered together, since they store highly similar information. In each case they record zero or more elements which should be in the given container, and zero or more elements which should not be in the container. As

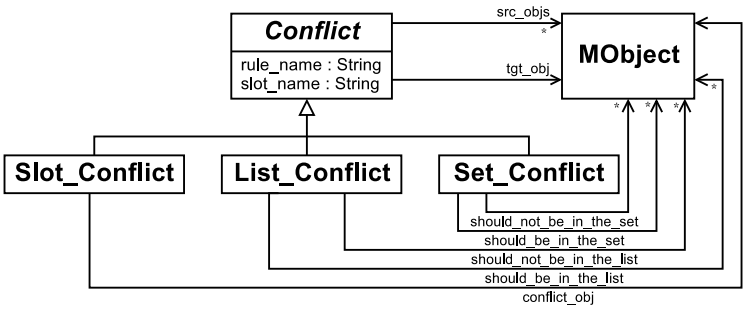


Figure 17: Conflict report model.

explained in section 3.7.1, at the time a conflict record is generated the list of elements which should not be in the container is only tentative; PMT and TM currently record all such elements, but dynamically filter them out when required to display conflict information.

7. Future work

Given its inherently experimental nature, PMT raises many questions and challenges for further work. As part of this, several engineering issues will need to be addressed before real-world usage is a possibility. Such issues include devising a practical mechanism for creating target identifiers that is more robust than the current string concatenation method, and so on. However I believe that once engineering issues are put to one side, two higher-level challenges are of particular interest.

The first is a relatively short term goal. PMT's approach to removing extraneous elements from the target model is often effective, but fails to remove elements if the links to those elements have existed for more than one round of change propagation. Since PMT uses the tracing information of the previous execution, if an element survives being removed in more than one round of change propagation, then PMT incorrectly assumes it has been manually added to the target model by the user. PMT can also, in some rarer cases, erroneously delete manually added links from the target model. Finding a practical means of accurately determining which elements can be safely removed from the target model would considerably improve the overall user experience of change propagation in PMT.

The second challenge I would consider to be a longer term goal, and relates to the efficiency of the approach. As explained in section 4.2, change propagation in PMT involves executing the whole transformation from the beginning. Whilst has the advantage that it can propagate even non-localised changes correctly, it is inevitably somewhat slow. On the other hand, approaches like Johann and Egyed optimise change propagation, but at the considerable expense of correctness. I believe that PMT's approach is a necessary 'fall back' option, but that there are two ways that may allow PMT to execute only a subset of the transformation in some cases. The first mechanism is directly influenced by Johann and Egyed. It may be possible to perform detailed analysis of some transformation rules, since model element patterns and model element expressions not containing arbitrary Converge code are effectively declarative statements relating two models. In such cases, it may then be possible to use this knowledge to determine that certain small changes only affect certain rules. The second mechanism may be complementary to the first: often the user will know whether certain of their transformations will be involved in the propagation of certain changes. If the user knows that certain types of changes are the ones most frequently propagated, they may be willing to 'mark up' parts of the transformation to indicate that certain paths need not be taken or, alternatively,

that certain paths must be taken, in the context of specific changes. I believe that working out appropriate analyses, and also practical mechanisms for 'marking up' a transformation for change propagation are considerable, but highly worthwhile, challenges.

8. Summary

In this paper I presented the PMT change propagating model transformation language. I started the paper by examining in more depth some of the issues, and design decisions, facing any change propagating model transformation approach. The motivating use case for PMT – allowing the user to manually alter the target model, whilst still allowing changes to be propagated into the altered model non-destructively – is important in understanding several of PMT's design decisions. I then presented PMT itself, exploring its approach to change propagation by example. PMT was shown to be capable of propagating even non-localised changes correctly. This led to an identification of some areas where PMT's change propagation techniques were effective, and some areas where they fell short of what one may wish for.

Despite its immaturity – particularly in comparison to MT upon which it is based – I believe that PMT is among the very first change propagating model transformation approaches to make a genuine attempt at exploring techniques for facilitating likely real-world scenarios. Although it can by no means be considered to be production ready in its current form, I believe it provides a basis for further exploration of this challenging and exciting area.

A. PMT grammar

PMT's grammar is identical to MT's with the exception of the `pt_mep_pattern` rule whose updated definition is as follows:

```
pt_mep_pattern ::= "(" "ID" ")" "[" "ID" pt_mep_pattern_op expr { "," "ID"
                   pt_mep_pattern_op expr }* "]"
               ::= "(" "ID" ")" "[" "]"
pt_mep_pattern_op ::= ":="
                  ::= "=="
                  ::= ":" "=="
                  ::= "!="
                  ::= ">="
                  ::= ":" ">="
                  ::= "<="
```

B. Model serializer

B.1. Overview

The TM `Serializer` module comprises functions to serialize and deserialize TM models, and tracing information. The serializer is essentially a simple graph walking function which flattens a model into an XML tree structure; references between nodes are made by using model elements' identifiers and an XML attribute `id`.

The deserializer is slightly more complex in operation. It utilizes Converge's `XML.Whole_Parser` module which provides a simple mechanism for parsing and traversing an XML file. The problem the deserializer faces is that as it works through its input creating appropriate model elements, it may find an `id` reference to an element which has not yet been created. In such cases, it creates a blank TM model element which it uses as a dummy holder to be filled in later when the full definition of the element is encountered in the file. This however means that during the process of deserialization the model being created may not be conformant to its meta-model. In order to prevent exceptions being raised whilst the model is deserialized, the deserializer sets the `_is_initialized` field of each element to `0`, ensuring that checks against the meta-model are not made. When all elements are completely deserialized, it then goes back over each element, setting this field to `1`, finally running the meta-models constraints against the meta-model to ensure that it has been recreated correctly.

B.2. Example output

This section shows the XML output from the TM `Serializer` model on the example of section 3.2. Firstly the ML2 input model:

```
<Model>
  <Element id="13" of="ML2_Package">
    <Attribute name="name">
      <String val="Personnel" />
    </Attribute>
    <Attribute name="elements">
      <Set>
        <Ref ref="12" />
        <Ref ref="11" />
        <Ref ref="10" />
      </Set>
    </Attribute>
  </Element>
  <Element id="12" of="ML2_Association">
    <Attribute name="name">
      <String val="PE" />
    </Attribute>
    <Attribute name="end2_name">
      <String val="manager" />
    </Attribute>
    <Attribute name="end1_name">
      <String val="employees" />
    </Attribute>
    <Attribute name="end2_multiplicity">
      <Int val="1" />
    </Attribute>
    <Attribute name="end1_multiplicity">
      <Int val="-1" />
    </Attribute>
```

```
    <Attribute name="end2_directed">
      <Int val="0" />
    </Attribute>
    <Attribute name="end1_directed">
      <Int val="0" />
    </Attribute>
    <Attribute name="end2">
      <Ref ref="11" />
    </Attribute>
    <Attribute name="end1">
      <Ref ref="10" />
    </Attribute>
  </Element>
  <Element id="11" of="ML2_Class">
    <Attribute name="name">
      <String val="Manager" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
  <Element id="10" of="ML2_Class">
    <Attribute name="name">
      <String val="Employee" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
</Model>
```

Then the ML1 target model produced by the transformation on its initial execution:

```
<Model>
  <Element id="Package_To_Package_0__13" of="ML1_Package">
    <Attribute name="name">
      <String val="Personnel" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
    <Attribute name="elements">
      <Set>
        <Ref ref="Association_To_Association_0__12" />
        <Ref ref="Association_To_Association_1__12" />
        <Ref ref="Class_To_Class_0__11" />
        <Ref ref="Class_To_Class_0__10" />
      </Set>
    </Attribute>
  </Element>
  <Element id="Association_To_Association_0__12" of="ML1_Association">
    <Attribute name="name">
      <String val="manager" />
    </Attribute>
    <Attribute name="multiplicity">
      <Int val="1" />
```

```
    </Attribute>
    <Attribute name="to">
      <Ref ref="Class_To_Class_0__11" />
    </Attribute>
    <Attribute name="from">
      <Ref ref="Class_To_Class_0__10" />
    </Attribute>
  </Element>
  <Element id="Association_To_Association_1__12" of="ML1_Association">
    <Attribute name="name">
      <String val="employees" />
    </Attribute>
    <Attribute name="multiplicity">
      <Int val="-1" />
    </Attribute>
    <Attribute name="to">
      <Ref ref="Class_To_Class_0__10" />
    </Attribute>
    <Attribute name="from">
      <Ref ref="Class_To_Class_0__11" />
    </Attribute>
  </Element>
  <Element id="Class_To_Class_0__11" of="ML1_Class">
    <Attribute name="name">
      <String val="Manager" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
  <Element id="Class_To_Class_0__10" of="ML1_Class">
    <Attribute name="name">
      <String val="Employee" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
</Model>
```

And finally the tracing information generated by the transformation on its initial execution:

```
<Tracing>
  <Trace rule="Class_To_Class">
    <From>
      <Ref ref="10" />
    </From>
    <To>
      <Ref ref="Class_To_Class_0__10" />
    </To>
  </Trace>
  <Trace rule="Class_To_Class">
    <From>
      <Ref ref="11" />
    </From>
    <To>
      <Ref ref="Class_To_Class_0__11" />
    </To>
```

```
  </Trace>
  <Trace rule="Association_To_Association">
    <From>
      <Ref ref="12" />
    </From>
    <To>
      <Ref ref="Association_To_Association_0__12" />
      <Ref ref="Association_To_Association_1__12" />
    </To>
  </Trace>
  <Trace rule="Package_To_Package">
    <From>
      <Ref ref="13" />
    </From>
    <To>
      <Ref ref="Package_To_Package_0__13" />
    </To>
  </Trace>
</Tracing>
```

References

[AP04] Marcus Alanen and Ivan Porres. Change propagation in a model-driven development tool. Presented at WiSME part of UML 2004, October 2004.

[BM03] Peter Braun and Frank Marschall. Botl – the bidirectional object oriented transformation language. Technical Report TUM-I0307, Institut für Informatik der Technischen Universität München, May 2003.

[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.

[CS03] Compuware and Sun. XMOF queries, views and transformations on models using MOF, OCL and patterns, August 2003. OMG document `ad/2003-08-07`.

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

[JE04] Sven Johann and Alexander Egyed. Instant and incremental transformation of models. September 2004.

[JL99] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1999.

[OJ04] Compuware. *OptimalJ*, 2004. `http://www.compuware.com/products/optimalj/`.

[RR93] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proc. 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510, 1993.

[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.

[Tra05a] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[Tra05b] Laurence Tratt. The MT model transformation language. Technical Report TR-05-02, Department of Computer Science, King's College London, May 2005.

[VV04] Gergely Varró and Dániel Varró. Graph transformation with incremental updates. In *Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques*, ENTCS, March 2004. To appear.