Electronic Communications of the EASST
Volume 29 (2010)

Proceedings of the
Ninth International Workshop on
Graph Transformation and
Visual Modeling Techniques
(GT-VMT 2010)

De-/Re-constructing Model Transformation Languages

Eugene Syriani and Hans Vangheluwe

14 pages

Guest Editors: Jochen Küster, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

De-/Re-constructing Model Transformation Languages

Eugene Syriani and Hans Vangheluwe

McGill University, School of Computer Science, Montréal, Canada
{esyria,hv}@cs.mcgill.ca

Abstract: The diversity of today's model transformation languages makes it hard to compare their expressiveness and provide a framework for interoperability. De-constructing and then re-constructing model transformation languages by means of a unique set of most primitive constructs facilitates both. We thus introduce **T-Core**, a collection of primitives for model transformation. Combining **T-Core** with a (programming or modelling) language enables the design of model transformation formalisms. We show how basic and more advanced features from existing model transformation languages can be re-constructed using **T-Core** primitives.

Keywords: Transformation primitives, multi-paradigm model transformation

1 Introduction

A plethora of different rule-based model transformation languages and supporting tools exist today. They cover all (or a subset of) the well-known essential features of model transformation [SV09c]: *atomicity*, *sequencing*, *branching*, *looping*, *non-determinism*, *recursion*, *parallelism*, *back-tracking*, *hierarchy*, and *time*. For such languages, the semantics (and hence implementation) of a transformation rule consists of the appropriate combination of building blocks implementing primitive operations such as matching, rewriting, and often a validation of consistent application of the rule. The abovementioned essential features of transformation languages are achieved by implicitly or explicitly specifying "rule scheduling". Languages such as **ATL** [JK06], **FUJABA** [FNTZ00], **GReAT** [AKK$^+$06], **MoTif** [SV09b], **VIATRA** [VB07], and **VMTS** [LLMC06] include constructs to specify the order in which rules are applied. This often takes the form of a control flow language. Without loss of generality, we consider transformation languages where models are encoded as typed, attributed graphs.

The diversity of transformation languages makes it hard, on the one hand, to compare their expressiveness and, on the other hand, to provide a framework for interoperability (*i.e.*, meaningfully combining transformation units specified in different transformation languages). One approach is to express model transformation at the level of primitive building blocks. De-constructing and then re-constructing model transformation languages by means of a small set of most primitive constructs offers a common basis to compare the expressiveness of transformation languages. It may also help in the discovery of novel, possibly in domain-specific, model transformation constructs by combining the building blocks in new ways. Furthermore, it allows implementers to focus on maximizing the efficiency of the primitives in isolation, leading to more efficient transformations overall. Lastly, once re-constructed, different transformation languages can seamlessly interoperate as they are built on the same primitives. This use of common primitives in turn allows for global as well as inter-rule optimization.

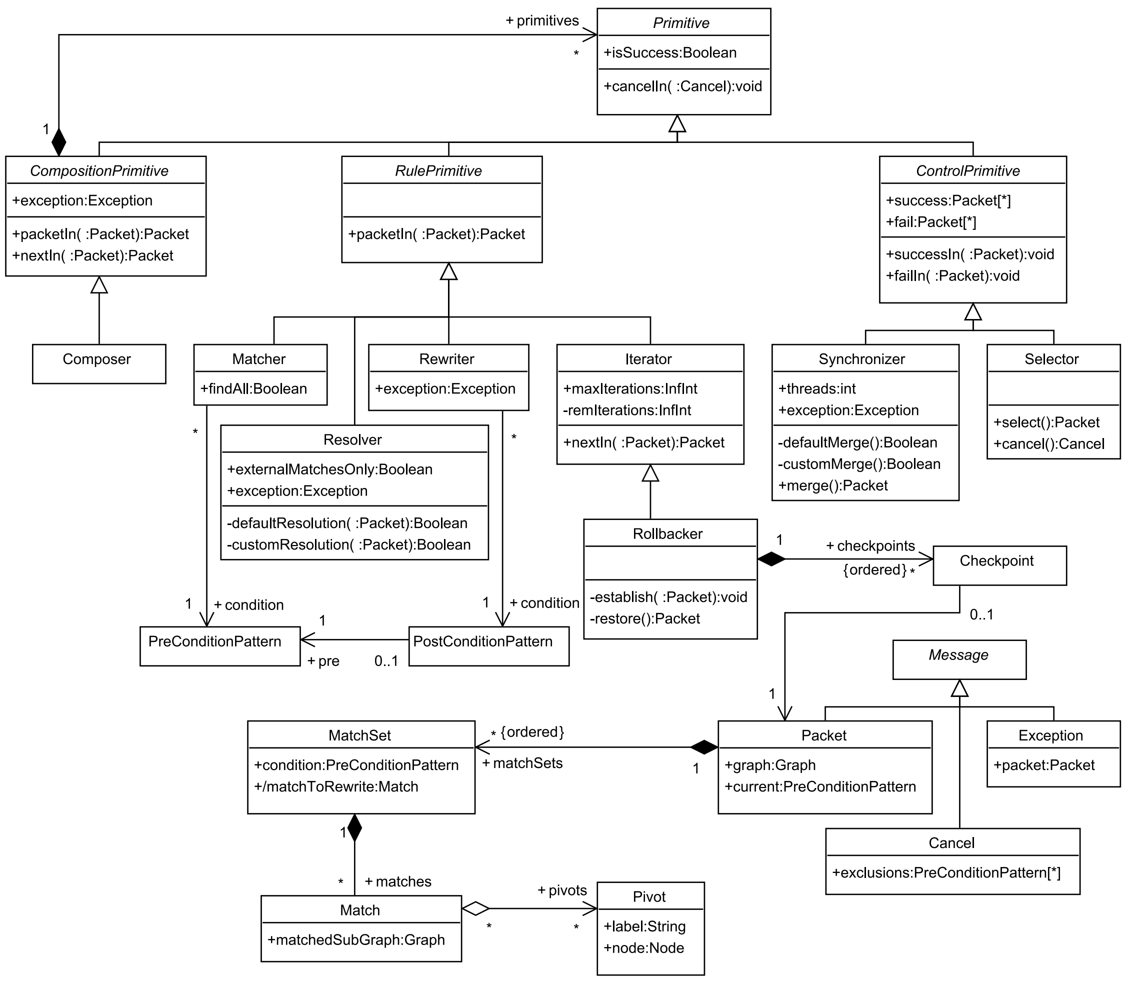

Figure 1: The **T-Core** module

We introduce **T-Core**, a collection of transformation language primitives for model transformation in Section 2. Section 3 motivates the choice of its primitives. Section 4 shows how transformation entities, common as well as more esoteric, can be re-constructed. Section 5 describes related work and Section 6 draws conclusions and presents directions for future work.

2 De-constructing Transformation Languages

We propose here a collection of model transformation primitives. The class diagram in Figure 1 presents the module **T-Core** encapsulating model transformation primitives. **T-Core** consists of eight primitive constructs (Primitive objects): a Matcher, Iterator, Rewriter, Resolver, Rollbacker, Composer, Selector, and Synchronizer. The first five are RulePrimitive elements and represent the building blocks of a single transformation unit. **T-Core** is not restricted to any form of specification of a transformation unit. In fact, we consider only PreConditionPatterns and PostConditionPatterns. For example, in rule-based model transformation, the transformation unit is a *rule*. The PreConditionPattern determines its applicability: it is usually described with a left-hand side (LHS) and optional negative application conditions (NACs). It also consists of a PostCondition-Pattern which imposes a pattern to be found after the rule was applied: it is usually described with a right-hand side (RHS). RulePrimitives are to be distinguished from the ControlPrimitives, which are used in the design of the rule scheduling part of the transformation language. A meaningful composition of all these different constructs in a Composer object allows modular encapsulation of and communication between Primitive objects.

Primitives exchange three different types of messages: Packet, Cancel, and Exception. A packet π represents the host model together with sufficient information for inter- and intra-rule processing of the matches. π thus holds the current model (graph in our case) graph, the match-Set, and a reference to the current PreConditionPattern identifying a MatchSet. A MatchSet refers to a condition pattern and contains the actual matches as well as a reference to the match-ToRewrite. Note that each MatchSet of a packet has a unique condition, used for identifying the set of matches. A Match consists of a sub-graph of the graph in π where each element is bound to an element in graph. Some elements (Nodes) of the match may be labelled as pivots, which allows certain elements of the model to be identified and passed between rules. A cancel message φ is meant to cancel the activity of an active primitive element (especially used in the presence of a Selector). Finally, specific exceptions χ can be explicitly raised, carrying along the currently processed packet π (π_ϕ is used to represent the empty packet).

All the primitive constructs can receive packets by invoking either their packetIn, nextIn, successIn, or failIn methods. The result of calling one of these methods sets the primitive in success or failure mode as recorded by the isSuccess attribute. Cancel messages are received from the cancelIn method. Next, we describe in detail the behaviour of the different methods supported by the primitive elements. A complete description can be found in [SV09a].

2.1 Matcher

The Matcher finds all possible matches of the condition pattern on the graph embedded in the packet it receives from its packetIn method. The transformation modeller may optimize the matching by setting the findAll attribute to false when he a priori knows that at most one match of this matcher will be processed in the overall transformation. The matching also considers the pivot mapping (if present) of the current match of π. After matching the graph, the Matcher stores the different matches in the packet as described in Algorithm 1. Some implementations may, for example, parametrize the Matcher by the condition pattern or embed it directly in the Matcher. The transformation units (*e.g.*, rules) may be compiled in pre/post-condition patterns or interpreted, but this is a tool implementation issue which we do not discuss here.

Algorithm 1 Matcher.packetIn(π)

```
$M \leftarrow$ (all) matches of condition found in $\pi$.graph
if $\exists\langle$condition$, M'\rangle \in \pi$.matchSets then
    $M' \leftarrow M' \cup M$
else
    add $\langle$condition$, M\rangle$ to $\pi$.matchSets
end if
$\pi$.current $\leftarrow$ condition
isSuccess $\leftarrow M \neq \emptyset$
return $\pi$
```

2.2 Rewriter

As described in Algorithm 2, the Rewriter applies the required transformation for its condition on the match specified in the packet it receives from its packetIn method. That match is consumed by the Rewriter: no other operation can be further applied on it. Some validations are made in the Rewriter to verify, for example, that π.current.condition = condition.pre or that no error occurred during the transformation. In our approach, a modification (update or delete) of an element in $\{M | \langle \text{condition}.pre, M\rangle \in \pi.\text{matchSets}\}$ is automatically propagated to the other matches, if applicable.

Algorithm 2 Rewriter.packetIn(π)

```
if $\pi$ is invalid then
    isSuccess $\leftarrow$ false
    exception $\leftarrow \chi(\pi)$
    return $\pi$
end if
$M \leftarrow \langle$condition.pre$, M\rangle \in \pi$.matchSets
apply transformation on $M$.matchToRewrite
if transformation failed then
    isSuccess $\leftarrow$ false
    exception $\leftarrow \chi(\pi)$
    return $\pi$
end if
set all modified nodes in $M$ to dirty
remove $\langle$condition$, M\rangle$ from $\pi$.matchSets
isSuccess $\leftarrow$ true
return $\pi$
```

2.3 Iterator

The Iterator chooses a match among the set of matches of the current condition of the packet it receives from its packetIn method, as described in Algorithm 3. The match is chosen randomly in a Monte-Carlo sense, repeatable using sampling from a uniform distribution to provide a reproducible, fair sampling. When its nextIn method is called, the Iterator chooses another match as long as the maximum number of iterations maxIterations (possibly infinite) is not yet reached, as described in Algorithm 4. In the case of multiple occurrences of a MatchSet identified by π.current, the Iterator selects the last MatchSet.

Algorithm 3 Iterator.packetIn(π)

```
if $\langle\pi$.current$, M\rangle \in \pi$.matchSets then
    choose $m \in M$
    $M$.matchToRewrite $\leftarrow m$
    remIterations $\leftarrow$ maxIterations $- 1$
    isSuccess $\leftarrow$ true
    return $\pi$
else
    isSuccess $\leftarrow$ false
    return $\pi$
end if
```

Algorithm 4 Iterator.nextIn(π)

```
if $\langle\pi$.current$, M\rangle \in \pi$.matchSets and remIterations $> 0$ then
    choose $m \in M$
    $M$.matchToRewrite $\leftarrow m$
    remIterations $\leftarrow$ remIterations $- 1$
    isSuccess $\leftarrow$ true
    return $\pi$
else
    isSuccess $\leftarrow$ false
    return $\pi$
end if
```

2.4 Resolver

The Resolver resolves a potential conflict between matches and rewritings as described in Algorithm 5. For the moment, the Resolver detects conflicts in a simple conservative way: it prohibits any change to other matches in the packet (check for *dirty* nodes). However, it does not verify if a modified match is still valid with respect to its pre-condition pattern. The externalMatchesOnly attribute specifies whether the conflict detection should also consider matches from its match set identified by π.current or not. In the case of conflict, a default resolution function is provided but the user may also override it.

Algorithm 5 Resolver.packetIn(π)

```
for all condition $c \in \{c | \langle c, M\rangle \in \pi$.matchSets$\}$ do
    if externalMatchesOnly and $c = \pi$.current then
        continue
    end if
    for all match $m \in M$ do
        if $m$ has a dirty node then
            if customResolution($\pi$) then
                isSuccess $\leftarrow$ true
                return $\pi$
            else if defaultResolution($\pi$) then
                isSuccess $\leftarrow$ true
                return $\pi$
            else
                isSuccess $\leftarrow$ false
                exception $\leftarrow \chi(\pi)$
                return $\pi$
            end if
        end if
    end for
end for
isSuccess $\leftarrow$ false
exception $\leftarrow \chi(\pi)$
return $\pi$
```

2.5 Rollbacker

The Rollbacker is only used to provide back-tracking capabilities to its transformation rule. Consequently, it is used as a recovery point that allows backward recovery of packets, *e.g.*, by means of checkpointing. The packetIn method establishes a checkpoint of the received packet and the nextIn method restores the last checkpoint to roll-back the packet to its previous state. Again, a maximum number of iterations can be specified.

Algorithm 6 Rollbacker.packetIn(π)

```
establish($\pi$)
remIterations $\leftarrow$ maxIterations $- 1$
isSuccess $\leftarrow$ **true**
**return** $\pi$
```

Algorithm 7 Rollbacker.nextIn(π)

```
**if** $\langle \pi$.current$, M \rangle \in \pi$.matchSets **and** remIterations $> 0$ **then**
    remIterations $\leftarrow$ remIterations $- 1$
    isSuccess $\leftarrow$ **true**
    **return** $\pi$
**else if** remIterations $> 0$ **then**
    $\hat{\pi} \leftarrow$ restore()
    remIterations $\leftarrow$ remIterations $- 1$
    isSuccess $\leftarrow$ **true**
    **return** $\hat{\pi}$
**else**
    isSuccess $\leftarrow$ **false**
    **return** $\pi$
**end if**
```

2.6 Selector

The Selector is used when a choice needs to be made between multiple packets processed concurrently by different constructs. It allows exactly one of them to be processed further. When its successIn (or failIn) method is called, the received packet is stored in its success (or fail) collection, respectively. Note that, unlike the previous described methods, it is only when the select method in Algorithm 8 is called that a packet is returned, chosen from success. The selection is random in the same way as in the Iterator. When the cancel method is invoked, the two collections are cleared and a cancel message φ is returned where the exclusions set consists of the singleton π.current (meaning that operations of the chosen condition should not be cancelled).

2.7 Synchronizer

The Synchronizer is used when multiple packets processed in parallel need to be synchronized. It is parametrized by the number of threads to synchronize. This number is known at design-time. Its successIn and failIn methods behave exactly like those of the Selector. The Synchronizer is in success mode only if all threads have terminated by never invoking failIn. The merge method "merges" the packets in success, as described in Algorithm 9. A trivial default merge function is provided by unifying and "gluing" the set of packets. Nevertheless, it first conservatively verifies the validity of the received packets by prohibiting overlapping matches between them. If it fails, the user can specify a custom merge function. This avoids the need for static parallel independence detection. Instead it is done at run-time and the transformation modeller must explicitly describe the handler. One pragmatic use of that solution is, for instance, to let the transformation run once to detect the possible conflicts and then the modeller may handle these cases by modifying the transformation model.

Algorithm 8 Selector.select()

```
if success $\neq \emptyset$ then
    $\hat{\pi} \leftarrow$ choose from success
    isSuccess $\leftarrow$ true
else if fail $\neq \emptyset$ then
    $\hat{\pi} \leftarrow$ choose from fail
    isSuccess $\leftarrow$ false
else
    $\hat{\pi} \leftarrow \pi_\phi$
    isSuccess $\leftarrow$ false
    exception $\leftarrow \chi(\pi_\phi)$
end if
success $\leftarrow \emptyset$
fail $\leftarrow \emptyset$
return $\hat{\pi}$
```

Algorithm 9 Synchronizer.merge()

```
if |success| = threads then
    if customMerge() then
        $\hat{\pi} \leftarrow$ the merged packet in success
        isSuccess $\leftarrow$ true
        success $\leftarrow \emptyset$
        fail $\leftarrow \emptyset$
        return $\hat{\pi}$
    else if defaultMerge() then
        $\hat{\pi} \leftarrow$ the merged packet in success
        isSuccess $\leftarrow$ true
        success $\leftarrow \emptyset$
        fail $\leftarrow \emptyset$
        return $\hat{\pi}$
    else
        isSuccess $\leftarrow$ false
        exception $\leftarrow \chi(\pi_\phi)$
        return $\pi_\phi$
    end if
else if |success| + |fail| = threads then
    $\hat{\pi} \leftarrow$ choose from fail
    isSuccess $\leftarrow$ false
    return $\hat{\pi}$
else
    isSuccess $\leftarrow$ false
    exception $\leftarrow \chi(\pi_\phi)$
    return $\pi_\phi$
end if
```

2.8 Composer

The Composer serves as a modular encapsulation interface of the elements in its primitives list. When one of its packetIn or nextIn methods is invoked, it is up to the user to manage subsequent method invocations to its primitives. Nevertheless, when the cancelIn method is called, the Composer invokes the cancelIn method of all its sub-primitives. This cancels the current action of the primitive object by resetting its state to its initial state. Cancelling happens only if it is actively processing a packet π such that the current condition of π is not in φ.exclusions, where φ is the received cancel message. In the case of a Matcher, since the current condition of the packet may not already be set, the cancelIn also verifies that the condition of the Matcher is not in the exclusions list. The interruption of activity can, for instance, be implemented as a pre-emptive asynchronous method call of cancelIn.

3 T-Core: a minimal collection of transformation primitives

In the de-construction process of transformation languages into a collection of primitives, questions like "up to what level?" or "what to include and what to exclude?" arise. The proposed **T-Core** module answers these questions in the following way.

In a model transformation language, the smallest transformation unit is traditionally the *rule*. A rule is a complex structure with a declarative part and an operational part. The declarative part of a rule consists of the specification of the rule (*e.g.*, LHS/RHS and optionally NAC in graph transformation rules). However, **T-Core** is not restricted to any form of specification let it be rule-based, constraint-based, or function-based. In fact, some languages require units with only a pre-condition to satisfy, while other with a pre- and a post-condition. Some even allow arbitrary permutations of repetitions of the two. In **T-Core**, either a PreConditionPattern or both a Pre- and a PostConditionPattern must be specified. For example, a graph transformation rule can be represented in **T-Core** as a couple of a pre- and a post-condition pattern, where the latter has a reference to the former to satisfy the semantics of the interface K (in the $L \leftarrow K \rightarrow R$ algebraic graph transformation rules) and be able to perform the transformation. Transformation languages where rules are expressed bidirectionally or as functions are supported in **T-Core** as long as they can be represented as pre- and post-condition patterns.

The operational part of a rule describes how it executes. This operation is often encapsulated in the form of an algorithm (with possibly local optimizations). Nevertheless, it always consists of a *matching phase*, *i.e.*, finding instances of the model that satisfy the pre-condition and of a *transformation phase*, *i.e.*, applying the rule such that the resulting model satisfies the post-condition. **T-Core** distinguishes these two phases by offering a Matcher and a Rewriter as primitives. Consequently, the Matcher's condition only consists of a pre-condition pattern and the Rewriter then needs a post-condition pattern that can access the pre-condition pattern to perform the rewrite. Combinations of Matchers and Rewriters in sequence can then represent a sequence of simple graph transformation rules: *match-rewrite-match-rewrite*. Moreover, because of the separation of these two phases, more general and complex transformation units may be built, such as: *match-match-match* or *match-match-rewrite-rewrite*. The former is a query where each Matcher filters the conditions of the query. The latter is a nesting of transformation rules. In this case, however, overlapping matches between different Matchers and then rewrites on the overlapping elements may lead to inconsistent transformations or even non-sense. This is why a Resolver can be used from **T-Core** to safely allow *match-rewrite* combinations.

The data structure exchanged between **T-Core** RulePrimitives in the form of packets contains sufficient information for each primitive to process it as described in the various algorithms in Section 2. The Match allows to refer to all model elements that satisfy a pre-condition pattern. The pivot mappings allow elements of certain matches to be bound to elements of previously matched elements. The pivot mapping is equivalent to passing parameters between rules as will be shown in the example in Section 4.1. The MatchSet allows to delay the rewriting phase instead of having to rewrite directly after matching.

Packets conceptually carry the complete model (optimized implementation may relax this) which allows concurrent execution of transformations. The Selector and the Synchronizer both permit to join branches or threads of concurrent transformations. Also, having separated the matching from the rewriting enables to manage the matches and the results of a rewrite by further operators. Advanced features such as iteration over multiple matches or back-tracking to a previous state in the transformation are also supported in **T-Core**.

Since **T-Core** is a low-level collection of model transformation primitives, combining its primitives to achieve relevant and useful transformations may involve a large number of these primitive operators. Therefore, it is necessary to provide a "grouping" mechanism. The Composer allows to modularly organize **T-Core** primitives. It serves as an interface to the primitives it encapsulates. This then enables scaling of transformations built on **T-Core** to large and complex model transformations designs.

T-Core is presented here as an open module which can be extended, through inheritance for example. One could add other primitive model transformation building blocks. For instance, a conformance check operator may be useful to verify if the resulting transformed model still conforms to its meta-model. It can be interleaved between sequences of rewrites or used at the end of the overall transformation as suggested in [KMS+09]. We believe however that such new constructs should either be part of the (programming or modelling) language or the tool in which **T-Core** is integrated, to keep **T-Core** as primitive as possible.

4 Re-constructing Transformation Languages

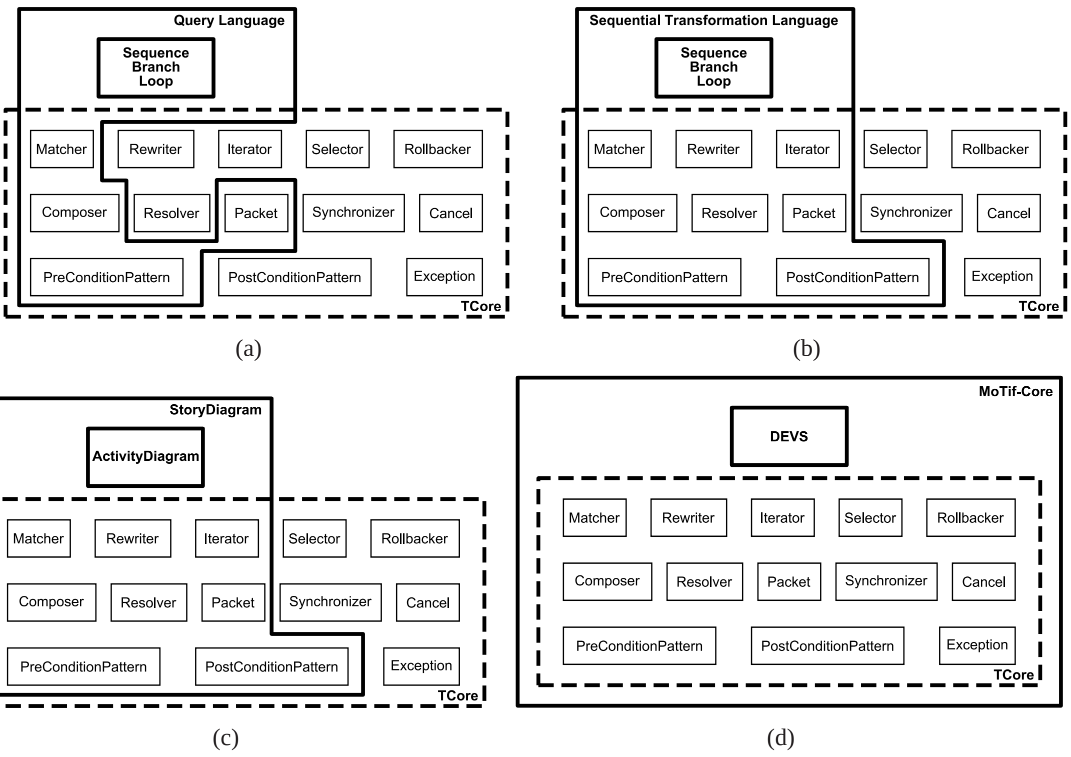

Figure 2: Combining **T-Core** with other languages allows to re-construct existing and new languages

Having de-constructed model transformation languages in a collection of model transformation primitives makes it easier to reason about transformation languages. In fact, properly combining **T-Core** primitives with an existing well-formed programming or modelling language allows us to re-construct some already existing transformation languages and even construct new ones [SV09a]. Figure 2 shows some examples of combinations of **T-Core** with other languages. Figure 2(a) and Figure 2(b) combine a subset of **T-Core** with a simple (programming) language which offers *sequencing*, *branching*, and *looping* mechanisms (as proposed in Böhm-Jocapini's *structured program theorem* [BJ66]). We will refer to such a language as an *SBL language*. The first combination only involves the Matcher and its PreConditionPattern, Packet messages to exchange, and the Composer to organize the primitives. These **T-Core** primitives integrated in an SBL language lead to a *query language*. Since only matching operations can be performed on the model, they represent queries where the resulting packet holds the set of all elements (sub-graph) of the model (graph) that satisfy the desired pre-conditions. Including other **T-Core** primitives such as the Rewriter promotes the query language to a transformation language. Figure 2(b) enumerates the necessary **T-Core** primitives combined with an SBL language to design a complete sequential model transformation language. Replacing the SBL language by another one, such as UML Activity Diagrams in Figure 2(c), allows us to re-construct Story Diagrams [FNTZ00], for example, since they are defined as a combination of UML Activity and Collaboration Diagrams with graph transformation features. Other combinations involving the whole **T-Core** module may lead to novel transformation language with exception handling and the notion of timed model transformations when combined with a discrete-event modelling language [SV09a].

We now present the re-construction of two transformation features using the combination of an SBL language with **T-Core** as in Figure 2(b).

4.1 Re-constructing Story Diagrams

In the context of object-oriented reverse-engineering, the **FUJABA** tool allows the user to specify the content of a class method by means of Story Diagrams. A Story Diagram organizes the behaviour of a method with activities and transitions. An activity can be a Story Pattern or a statement activity. The former consists of a graph transformation rule and the latter is Java code. Figure 3 shows such a story diagram taken from the doDemo method example in [FNTZ00]. This snippet represents an elevator loading people on a given floor of a house who wish to go to another level. The rule in the pattern is specified in a UML Collaboration Diagram-like notation with objects and associations. Objects with implicit types (*e.g.*, this, l2, and e1) are *bound* objects from previous patterns or variables in the context of the current method. The Story Pattern 6 is a for-all Pattern. Its rule is applied on all matches found looping over the unbound objects (*e.g.*, p4, and l4). The outgoing transition labelled each

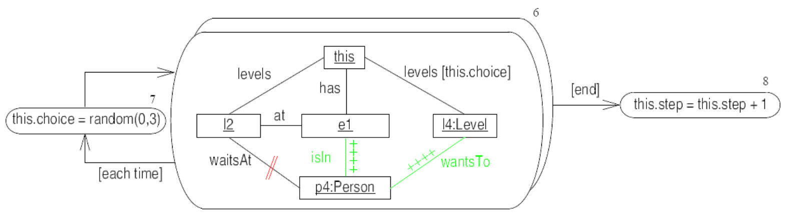

Figure 3: The **FUJABA** doSubDemo transformation showing a for-all Pattern and two statement activities

time applies statement 7 after each iteration of the for-all Pattern. This activity allows the pattern to simulate random choices of levels for different people in the elevator. When all iterations have been completed, the flow proceeds with statement 8 reached by the transition labelled end. The latter activity simulates the elevator going one level up.

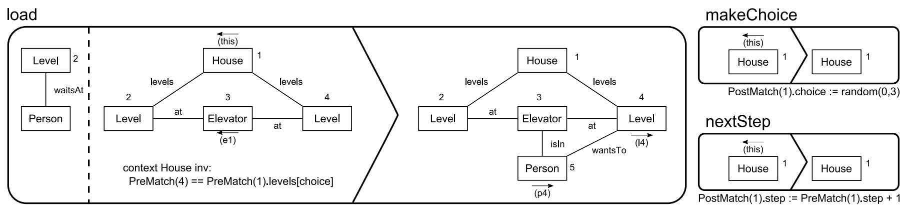

Figure 4: The three **MoTif** rules for the doSubDemo transformation

We now show how to re-construct this non-trivial story diagram transformation from an SLB language combined with **T-Core**. An instance of that combination is called a ***T-Core** model*. First, we design the rules needed for the conditions of rule primitives. Figure 4 describes the three necessary rules corresponding to the three Story Diagram activities. We use the syntax of **MoTif** [SV09b] where the central compartment is the LHS, the compartment on the right of the arrow head is the RHS and the compartment(s) on the left of dashed lines are the NAC(s). The concrete syntax for representing the pattern elements was chosen to be intuitively close enough to the **FUJABA** graphical representation. Numeric labels are used to uniquely identify different elements across compartments. Elements with an alpha-numeric label between parentheses denote pivots. A right-directed arrow on top of such a label depicts that the model element matched for this pattern element is assigned to a pivot (*e.g.*, p4 and l4). If the arrow is directed to the left, then the model element matched for this pattern element is bound to the specified pivot (*e.g.*, this and e1).

Algorithm 10 makeChoiceC.packetIn(π)

```
$\pi \leftarrow$ makeChoiceM.packetIn($\pi$)
if not makeChoiceM.isSuccess then
    isSuccess $\leftarrow$ false
    return $\pi$
end if
$\pi \leftarrow$ makeChoiceI.packetIn($\pi$)
if not makeChoiceI.isSuccess then
    isSuccess $\leftarrow$ true
    return $\pi$
end if
$\pi \leftarrow$ makeChoiceW.packetIn($\pi$)
if not makeChoiceW.isSuccess then
    isSuccess $\leftarrow$ false
    return $\pi$
end if
$\pi \leftarrow$ makeChoiceR.packetIn($\pi$)
if not makeChoiceR.isSuccess then
    isSuccess $\leftarrow$ false
    return $\pi$
end if
isSuccess $\leftarrow$ true
return $\pi$
```

The **T-Core** model equivalent to the original doSubDemo transformation consists of a Composer doSubDemoC. It is composed of two Composers loadC and nextStepC each containing a Matcher, an Iterator, a Rewriter, and a Resolver. The packetIn method of doSubDemoC first calls the corresponding method of loadC and then feeds the returned packet to the packetIn method of nextStepC. This ensures that the output packet of the overall transformation is the result of first loading all the Person objects and then moving the elevator by one step. makeChoiceC and nextStepC behave as simple transformation rules. Their packetIn method behaves as specified in Algorithm 10. First, the matcher is tried on the input packet. Note that the conditions of the matchers makeChoiceM and nextStepM are the LHSs of rules **makeChoice** and **nextStep**, respectively. If it fails, the composer goes into failure mode and the packet is returned. Then, the iterator chooses a match. Subsequently, the rewriter attempts to transform this match.

Note that the conditions of the rewriters makeChoiceW and nextStepW are the RHSs of rules **makeChoice** and **nextStep**, respectively. If it fails, an exception is thrown and the transformation stops. Otherwise, the resolver verifies the application of this pattern with respect to other matches in the transformed packet. The behaviour of the resolution function will be elaborated on later. Finally, on a successful resolution, the resulting packet is output and the composer is put in success mode. loadC is the composer that emulates the for-all Pattern of the example. Algorithm 11 specifies that behaviour. After finding all matches with loadM (whose condition is the LHS and the NAC of rule **load**), the packet is forwarded to the iterator loadI to choose a match. The iteration is emulated by a loop with the failure mode of loadI as the breaking condition. Inside the loop, loadW rewrites the chosen match and loadR resolves possible conflicts. Then, the resulting packet is sent to makeChoiceC to fulfil the each time transition of the story digram. After that, the nextIn method of loadI is invoked with the new packet to choose a new match and proceed in the loop.

Algorithm 11 loadC.packetIn(π)

```
$\pi \leftarrow$ loadM.packetIn($\pi$)
if not loadM.isSuccess then
    isSuccess $\leftarrow$ false
    return $\pi$
end if
$\pi \leftarrow$ loadI.packetIn($\pi$)
while true do
    if not loadI.isSuccess then
        isSuccess $\leftarrow$ true
        return $\pi$
    end if
    $\pi \leftarrow$ loadW.packetIn($\pi$)
    if not loadW.isSuccess then
        isSuccess $\leftarrow$ false
        return $\pi$
    end if
    $\pi \leftarrow$ loadR.packetIn($\pi$)
    if not loadR.isSuccess then
        isSuccess $\leftarrow$ false
        return $\pi$
    end if
    $\pi \leftarrow$ makeChoiceC.packetIn($\pi$)
    $\pi \leftarrow$ loadI.nextIn($\pi$)
end while
isSuccess $\leftarrow$ true
return $\pi$
```

Having seen the overall **T-Core** transformation model, let us inspect how the different Resolvers should behave in order to provide a correct and complete transformation. The first rewriter called is loadR and the first time it receives a packet is when a transformation is applied on one of the matches of loadM. Therefore each match consists of the same House (since it is a bound node), two Levels, an Elevator, and the associations between them. On the other hand, loadW only adds a Person and links it to a Level. Therefore the default resolution function of loadR applies successfully, since no matched element is modified nor is the NAC violated in any other match. The next resolver is makeChoiceR which is in the same loop as loadR. There, the House is conflicting with all the matches in the packet according to the conservative default resolution function. Note that makeChoiceM finds at most one match (the bound House element). However, makeChoiceW does not really conflict with matches found in loadM. We therefore specify a custom resolution function for makeChoiceR that always succeeds. The same applies for nextStepR.

4.2 Re-constructing amalgamated rules

In a recent paper, Rensink et al. claim that the *Repotting the Geraniums* example is inexpressible in most transformation formalisms [RK09]. The authors propose a transformation language that uses an amalgamation scheme for nested graph transformation rules. As we have seen in the previous example, nesting transformation rules is possible in **T-Core** and will be used to solve the problem. It consists of *repotting all flowering geraniums whose pots have cracked*. Figure 5 illustrates the two nested

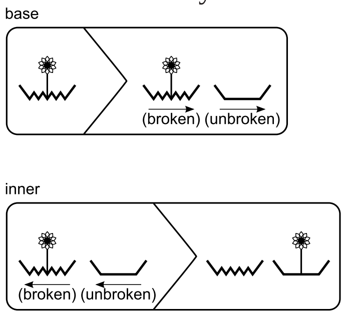

Figure 5: The transformation rules for the *Repotting Geraniums* example

graph transformation rules involved and Algorithm 12 demonstrates the composition of primitive **T-Core** elements encoding these rules. baseM (with, as condition, the LHS of rule **base**) finds all broken pots containing a flowering geranium, given the input packet containing the input graph. The set of matches resulting in the packet are the combination of all flowering geraniums and their pot container. From then on starts the loop. First, baseI chooses a match. If one is chosen, baseW transforms this match and baseR resolves any conflicts. In this case, baseW only creates a new unbroken pot and assigns pivots. Therefore, baseR's resolution function always succeeds. In fact, the resolver is not needed here, but we include it for consistency. The innerC composer encodes the **inner** rule which finds the two bound pots and moves a flourishing flower in the broken pot to the unbroken one. In order to iterate over all the flowers in the broken pot, the innerC.packetIn method has the exact same behaviour as loadC.packetIn in Algorithm 11, with the exception of not calling a sub-composer (like makeChoiceC). Note that an always successful custom resolution function for innerR is required. After the Resolver successfully outputs the packet, the **inner** rule is applied. Then (and also if baseI had failed) baseM.packetIn is called again with the resulting packet. The loop ends when the baseM.packetIn method call inside the loop fails, which entails baseC to return the final packet in success mode.

Algorithm 12 baseC.packetIn(π)

```
$\pi \leftarrow$ baseM.packetIn($\pi$)
if not baseM.isSuccess then
    isSuccess $\leftarrow$ false
    return $\pi$
end if
while true do
    $\pi \leftarrow$ baseI.packetIn($\pi$)
    if not baseI.isSuccess then
        isSuccess $\leftarrow$ true
        return $\pi$
    end if
    $\pi \leftarrow$ baseW.packetIn($\pi$)
    if not baseW.isSuccess then
        isSuccess $\leftarrow$ false
        return $\pi$
    end if
    $\pi \leftarrow$ baseR.packetIn($\pi$)
    if not baseR.isSuccess then
        isSuccess $\leftarrow$ false
        return $\pi$
    end if
    $\pi \leftarrow$ innerC.packetIn($\pi$)
    $\pi \leftarrow$ baseM.packetIn($\pi$)
    if not baseM.isSuccess then
        isSuccess $\leftarrow$ true
        return $\pi$
    end if
end while
```

5 Related work

The closer work to our knowledge is [VJBB09]. In the context of global model management, the authors define a type system offering a set of primitives for model transformation. The advantage of our approach is that **T-Core** is a described here as a module and is thus directly implementable. We have recently incorporated **T-Core** with an asynchronous and timed modelling language [SV09a] which allowed us to re-implement the two examples in Section 4 as well as others. Also, the approach described in [VJBB09], does not deal with exceptions at all. Nevertheless, their framework is able to achieve higher-order transformations, which we did not consider in this paper.

The **GP** graph transformation language [MP08] also offers transformation primitives. They however focus more on the scheduling of the rules then on the rules themselves. Their scheduling (control) language is an extension of an SBL language. Our approach is more general since much more complex scheduling languages (*e.g.*, allowing concurrent and timed transformation execution) can be integrated with **T-Core**. Although it performs very efficiently, the application area of **GP** is more limited, as it can not deal with arbitrary domain-specific models.

Other graph transformation tools, such as **VIATRA** [VB07] and **GReAT** [AKK+06], have their own virtual machine used as an API. In our approach, since the primitive operations are modelled, they are completely compatible with other existing model transformation frameworks.

6 Conclusion

In this paper, we have motivated the need for providing a collection of primitives for model transformation languages. We have defined **T-Core** which precisely describes each of these primitive constructs. The de-construction process of model transformation languages enabled us to re-construct existing model transformation features by combining **T-Core** with, for example, an SBL language. This allowed us to compare different model transformation languages using a common basis.

Now that these primitives are well-defined, efficiently implementing each of them might lead to more efficient model transformation languages. Also, for future work, we would like to investigate how **T-Core** combined with appropriate modelling languages can express further transformation constructs. We would also like to investigate further on the notion of exceptions and error handling in the context of model transformation.

Bibliography

[AKK+06] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *SoSym* 5(3):261–288, September 2006.

[BJ66] C. Böhm, G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM* 9(5):366–371, May 1966.

[FNTZ00] T. Fischer, J. Niere, L. Turunski, A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Ehrig et al. (eds.), *Theory and Application of Graph Transformations*. LNCS 1764, pp. 296–309. Springer-Verlag, Paderborn (Germany), November 2000.

[JK06] F. Jouault, I. Kurtev. Transforming Models with ATL. In *MTiP'05*. LNCS 3844, pp. 128–138. Springer-Verlag, January 2006.

[KMS+09] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer. Systematic Transformation Development. In *3rd International Workshop on Multi-Paradigm Modeling (best paper)*. Volume 21. October 2009.

[LLMC06] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf. Model Transformation with a Visual Control Flow Language. *IJCS* 1(1):45–53, 2006.

[MP08] G. Manning, D. Plump. The GP Programming System. In *GT-VMT'08*. ECEASST, pp. 235–247. Budapest (Hungary), March 2008.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. In Margaria et al. (eds.), *GT-VMT'09*. EASST. York (UK), March 2009.

[SV09a] E. Syriani, H. Vangheluwe. De-/Re-constructing Model Transformation Languages. Technical report SOCS-TR-2009.8, McGill University, School of Computer Science, August 2009.

[SV09b] E. Syriani, H. Vangheluwe. *Discrete-Event Modeling and Simulation: Theory and Applications*. Chapter DEVS as a Semantic Domain for Programmed Graph Transformation. CRC Press, Boca Raton (USA), 2009.

[SV09c] E. Syriani, H. Vangheluwe. Matters of model transformation. Technical report SOCS-TR-2009.2, McGill University, School of Computer Science, March 2009.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68(3):214–234, 2007.

[VJBB09] A. Vignaga, F. Jouault, M. C. Bastarrica, H. Brunelière. Typing in Model Management. In Paige (ed.), *Theory and Practice of Model Transformations (ICMT'09)*. LNCS 5563, pp. 197–212. Springer-Verlag, Zürich (Switzerland), June 2009.