**SoSym manuscript No.**
(will be inserted by the editor)

# T-Core: A Framework for Custom-built Transformation Languages

**Eugene Syriani · Brian LaShomb · Hans Vangheluwe**

**Abstract** A large number of model transformation languages and tools have emerged in the past decade. A transformation engineer is thus left with too many choices for the language he shall use to perform a specific transformation task. Furthermore, it is currently not possible to combine or re-use transformations implemented in different languages. We therefore propose T-Core, a framework where primitive transformation operators can be combined to define and encapsulate re-usable model transformation idioms. In this context the transformation engineer is free to use existing transformation building blocks from an extensible library or define his own transformation units. The proposed primitive transformation operators result from de-construction process of different existing transformation languages. Re-constructing these languages offers a common basis to compare the expressiveness, provides a framework for inter-operating them, and allows the transformation engineer to design transformations with the most appropriate constructs for his task.

**Keywords** model transformation · domain-specific model transformation · transformation library · re-engineering

## 1 Introduction

For the past decade, we have witnessed a plethora of different rule-based MTLs and supporting tools. This sudden overwhelming emergence is the result of

E. Syriani
University of Alabama, Tuscaloosa AL, U.S.A.
E-mail: esyriani@cs.ua.edu

B. LaShomb
University of Alabama, Tuscaloosa AL, U.S.A.
E-mail: lasho001@crimson.ua.edu

H. Vangheluwe
University of Antwerp, B-2020 Antwerp, Belgium
E-mail: hv@cs.mcgill.ca

not yet having "the best fit" language for transforming models. QVT [23] and ATL [26] are examples that attempt to reach this goal. The problem with these candidates is that they are general-purpose transformation languages that often encumber the developer with unneeded features. This adds complexity to the design the transformation which may lead to design errors and even reduce productivity [27]. There is thus a need to have model transformation languages (MTLs) that are specific to the problem to solve. Other more focused MTLs are dedicated to implement specific transformation solution [5] and some even restrict themselves to specific domains of application [9].

Many MTLs (FUJABA [17], GReAT [1], ProGReS [47], VIATRA [57], VMTS [36], etc.) have been de facto used in scenarios they were originally not intended for. In order to resolve the issues encountered, more features have been added to the languages to widen their scope. They cover all (or a subset of) the well-known essential features of model transformation [13,39,52]. 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 above-mentioned essential features of transformation languages are achieved by implicitly or explicitly specifying "rule scheduling". Languages include constructs to specify the order in which rules are applied, often in the form of a control flow language.

The diversity of transformation languages makes it hard to (1) easily design complex transformations, (2) compare their expressiveness, and (3) 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. Deconstructing and then re-constructing MTLs 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 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 inter-operate as they are built on the same primitives. This use of common primitives in turn allows for global as well as inter-rule optimization.

The vision behind this article is to provide the tools to develop domain-specific MTLs as opposed to general-purpose MTLs. In previous work [32,55], we have illustrated techniques to model the syntax of such languages. The main contribution of this article is a framework that offers model transformation operators that can be mixed and matched in order to design transformations with no more expressiveness and computation power than needed by the user. Such primitives give a uniform semantics in terms of the computation of the transformation as well as the expressiveness of its constructs *i.e.,* the *transformation units*. Transformation units, as first introduced in [28], are meant to "divide a large set of transformation rules into smaller ones in a structured and system-

| Model Transformation Paradigm | Object-Oriented Paradigm |
|---|---|
| Transformation | Package |
| Rule | Class |
| Rule Primitive | Method |
| CRUD operation | Operation on variables |

**Table 1** Analogy of the abstraction hierarchy in model transformation and object-oriented paradigms.

atic way". Typical transformation units are rules, queries, their composition, and helper functions that are re-usable in different model transformations.

In Section 2 we introduce T-Core, a collection of primitive operators for MTL. Section 3 addresses performance issues of the main T-Core operators. Then, in Section 4, we describe a framework for engineering a product line of problem-specific MTLs based on T-Core. As it is unfeasible to validate the completeness of the collection, we show by example how transformation entities, common as well as more esoteric, can be re-constructed using T-Core. Section 5 outlines a complete application of the proposed framework by entirely re-engineering an existing MTL. Finally Section 6 discusses related work and we conclude in Section 7.

## 2 A Minimal Transformation Core

Model transformation language primitives can be defined at different levels of granularity. The decomposition process is similar to what is found in object-oriented languages as depicted in Table 1. At the highest level, the transformation can be decomposed into sub-transformations[1], each dedicated to a specific task in order to accomplish a single goal (simulation, code generation, synchronization, etc). Following the analogy, a transformation corresponds to a package in object-oriented languages. Defining MTL primitives at this level means that transformations are treated as black-boxes, which is not the intention. Thus at a lower level, a (sub-)transformation can be decomposed into individual rules. Rules are the units of a transformation like a class is to a package. However, setting the rules as primitives would not consider other model transformation paradigms such as relational or functional. At a coarser level of abstraction, a transformation encapsulates CRUD operations performed on a model. However, we believe that these operations should be defined at the virtual machine level, rather than having a transformation language engineer combine them, *i.e.,* this is not the optimal level of abstraction. Hence rule primitives reside somewhere between rule definitions and CRUD operations. They dictate how a rule operates. As methods define the behavior of a class

[1] A sub-transformation can be considered as a transformation on its own. But when designed modularly, composing these transformations can lead to a more complex transformation.

and operations on objects, rule primitives define the behavior of a rule operating on the model. At a more fine-grained level, a rule primitive encapsulates CRUD operations performed on the model, such as common transformation virtual machines [3]. This is similar to how methods encapsulate operations that can be performed on variables (assignment, navigation, iteration, etc).

The proposed decomposition of MTLs therefore focuses on the rule primitives level. In previous work [49], we have compared the features of thirteen MTLs. One can synthesize the common essential features of model transformation as follows:

- **Pre- and post-condition patterns** that allow one to declaratively specify a rule;
- **Matching** rule pre-condition patterns in the host model to bind model elements (a match) that will be modified by the application of a rule;
- **Rewriting** the host model to satisfy the post-condition of a rule;
- **Validation** of consistent rule applications to detect conflicts and resolve them;
- **Manipulation of matches** to **iterate** through them and **roll-back** to previous match states;
- **Control of the flow** of rule applications by offering **choices** and **concurrency**;
- **Composition** mechanisms to provide structure, re-use, and encapsulation.

Based on the previous observations, we propose here a collection of model transformation primitives. The class diagram in Fig. 1 presents the module T-Core (which stands for Transformation 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). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. It also consists of a PostConditionPattern which imposes a pattern to be found after the rule was applied: it is usually described with a 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 $\pi$ represents the host model together with sufficient information for inter- and intra-rule processing of the matches. $\pi$ thus holds the current model (a typed, attributed graph in our case) `graph`, the `matchSet`,

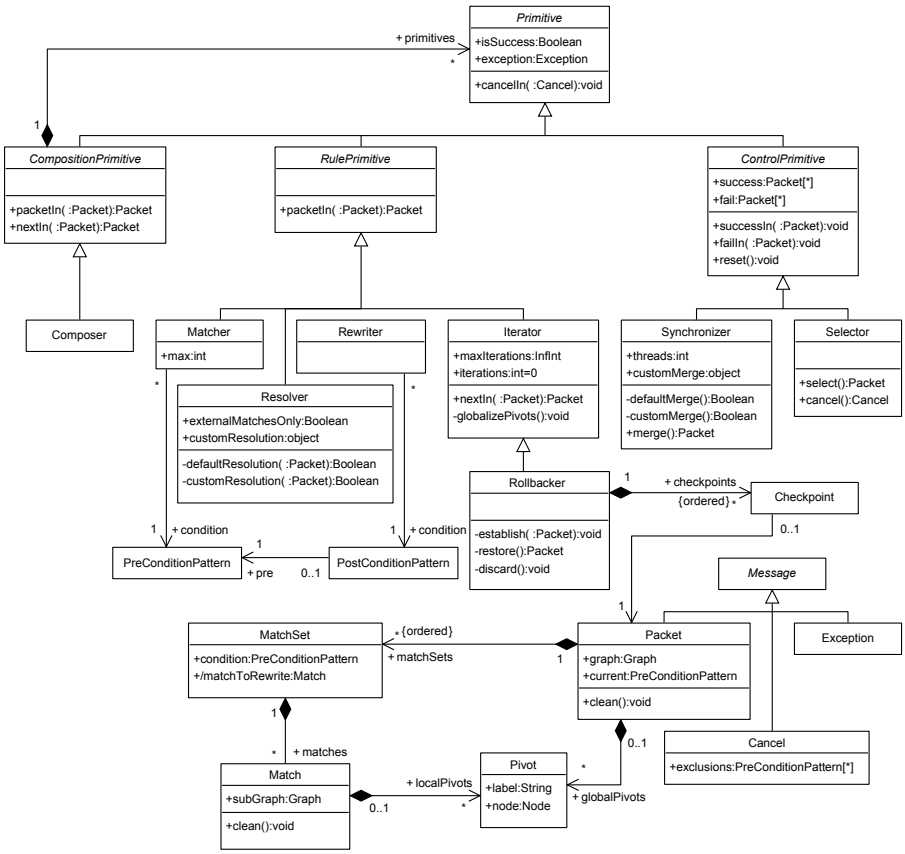



**Fig. 1** The T-Core module.

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 `matchToRewrite`. 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 $\pi$ 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 $\varphi$ is meant to cancel the activity of an active primitive element (especially used in the presence of a Selector). Finally, specific exceptions $\chi$ can be explicitly raised, carrying along the currently processed packet $\pi$. More details on how transformation exceptions are handled can be found in [51].

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.

## 2.1 The Primitives

Here we describe in detail the behaviour of the different methods supported by each of the eight primitive elements.

### *2.1.1 Matcher*

**Algorithm 1** Matcher.`packetIn`($\pi$)

```
1: M ← match(π.graph, condition, π.globalPivots)
2: if ∃⟨condition, M'⟩ ∈ π.matchSets then
3:     M' ← M' ∪ M
4: else
5:     add ⟨condition, M⟩ to π.matchSets
6: π.current ← condition
7: isSuccess ← M ≠ ∅
8: return π
```

The Matcher looks for an occurrence of its pre-condition pattern `condition` in the graph of the input packet $\pi$. The details of the `match` procedure is given in Algorithm 12 in Section 3.3. The transformation modeler may optimize the matching by setting the `max` attribute to finding one, all, or a maximum number of matches when he knows a priori that this many matches of the matcher will be processed in the overall transformation. The matching also considers the pivot mapping[2] (if present) of the current match of $\pi$. After matching the graph, the Matcher stores the different matches in the packet as described in Algorithm 1. In this notation, $MS$ is a MatchSet object structure, $M$ is the set of Match instances it holds and $m$ is a single Match object. 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 is not discussed here. The complexity of the Matcher depends on that of the matching algorithm in line 1 which we discuss in the next section.

### *2.1.2 Rewriter*

As described in Algorithm 2, the Rewriter applies the required transformation according to the post-condition pattern `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 $\pi$.`current.condition` = `condition.pre` or that no error occurred during the transformation. In our approach, a modification (update or delete) of an element in $\{m \in M | \langle \texttt{condition}.pre, M\rangle \in \pi.\texttt{matchSets}\}$ is automatically propagated to all the

[2] The bound pivot nodes are stored in `globalPivots`. But the matching may also assign pivots (useful for nested rules, as discussed later) and stores them in `localPivots`.

**Algorithm 2** Rewriter.`packetIn(`$\pi$`)`

```
1: if π is invalid then
2:     isSuccess ← false
3:     exception ← χ(π)
4:     return π
5: MS ← ⟨condition.pre, M⟩ ∈ π.matchSets
6: apply transformation on MS.matchToRewrite
7: if transformation failed then
8:     isSuccess ← false
9:     exception ← χ(π)
10:    return π
11: set all modified nodes in MS.matchToRewrite to dirty
12: remove MS.matchToRewrite from MS.matches
13: isSuccess ← true
14: return π
```

other matches, when applicable. The complexity of the Rewriter depends on that of the rewriting algorithm in line 6 which we discuss in the next section.

### *2.1.3 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 $\pi$.`current`, the Iterator selects the last MatchSet. The complexity of the Iterator is constant with an appropriate pseudo-random number generator for choosing a match in line 3.

**Algorithm 3** Iterator.`packetIn(`$\pi$`)`

```
1: if ⟨π.current, M⟩ ∈ π.matchSets then
2:     MS ← ⟨π.current, M⟩
3:     choose m ∈ MS.matches
4:     MS.matchToRewrite ← m
5:     iterations ← 1
6:     isSuccess ← true
7:     return π
8: else
9:     isSuccess ← false
10:    return π
```

**Algorithm 4** Iterator.`nextIn(`$\pi$`)`

```
1: if ⟨π.current, M⟩ ∈ π.matchSets and
   iterations < maxIterations then
2:     MS ← ⟨π.current, M⟩
3:     choose m ∈ MS.matches
4:     MS.matchToRewrite ← m
5:     iterations ← iterations + 1
6:     isSuccess ← true
7:     return π
8: else
9:     isSuccess ← false
10:    return π
```

### *2.1.4 Resolver*

The Resolver resolves a potential conflict between matches and rewritings as described in Algorithm 5. For the moment, the Resolver detects conflicts

**Algorithm 5** Resolver.`packetIn(`$\pi$`)`

```
for all condition c ∈ {c| ⟨c, M⟩ ∈ π.matchSets} do
  if externalMatchesOnly and c = π.current then
    continue
  for all match m ∈ M do
    if m has a dirty node then
      if not customResolution(π) then
        if not defaultResolution(π) then
          isSuccess ← false
          exception ← χ(π)
          return π
isSuccess ← true
return π
```

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 $\pi$.`current` or not. In the case of conflict, a default resolution function is provided but the user may also override it. Although the conflict detection is conservative, the `customResolution` function may discard the conflict if, for example, NACs are not enabled in other matches. That is, the Resolver will detect trivial conflicts, but the transformation engineer is empowered to define the conflicts that may occur in his application domain. The complexity of the Resolver depends on the implementation of the custom resolution function as well as the number of match sets and matches. Note that the complexity of the default resolution function is linear in terms of the size of the match.

### *2.1.5 Rollbacker*

**Algorithm 6** Rollbacker.`packetIn(`$\pi$`)`

```
establish(π)
if ⟨π.current, M⟩ ∈ π.matchSets then
  maxIterations ← |M|
else
  maxIterations ← max
iterations ← 1
isSuccess ← true
return π
```

**Algorithm 7** Rollbacker.`nextIn(`$\pi$`)`

```
π̂ ← restore()
iterations ← iterations + 1
if iterations < maxIterations then
  isSuccess ← true
else
  discard()
  isSuccess ← false
return π̂
```

The Rollbacker provides transactional behavior with back-tracking capabilities. Consequently, it is used as a recovery point that allows backward recovery of packets, *e.g.,* by means of checkpointing as described in Algorithms 6 and 7. The `packetIn` method establishes a checkpoint of the received packet. This is done by making a copy $\hat{\pi}$ of the input packet $\pi$ and pushing it on a temporary stack. It also sets the maximum number of iterations to the total number of matches found for the current condition. The `nextIn` method restores the last

checkpoint to roll-back the packet to its previous state $\hat{\pi}$. If there are no more matches left in $M$, it also removes the previous checkpoint established. The transactional operations of the Rollbacker have a linear worst case complexity in terms of the size of the packet.

### *2.1.6 Selector*

**Algorithm 8** Selector.`select()`

```
1: if success ≠ ∅ then
2:     π̂ ← choose from success
3:     isSuccess ← true
4: else if fail ≠ ∅ then
5:     π̂ ← choose from fail
6:     isSuccess ← false
7: else
8:     π̂ ← π_φ
9:     isSuccess ← false
10:    exception ← χ(π_φ)
11: success ← ∅
12: fail ← ∅
13: return π̂
```

**Algorithm 9** Synchronizer.`merge()`

```
1: if |success| = threads then
2:     if customMerge() then
3:         π̂ ← the merged packet in success
4:         isSuccess ← true
5:         success ← ∅
6:         fail ← ∅
7:         return π̂
8:     else if defaultMerge() then
9:         π̂ ← the merged packet in success
10:        isSuccess ← true
11:        success ← ∅
12:        fail ← ∅
13:        return π̂
14:    else
15:        isSuccess ← false
16:        exception ← χ(π_φ)
17:        return π_φ
18: else if |success|+|fail| = threads then
19:    π̂ ← choose from fail
20:    isSuccess ← false
21:    return π̂
22: else
23:    isSuccess ← false
24:    exception ← χ(π_φ)
25:    return π_φ
```

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 previously 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. However, if `success` is empty, the returned packet is randomly chosen from `fail`. Note that if both `success` and `fail` are empty, `select` throws an exception with an empty packet $\pi_\phi$. When the `cancel` method is invoked, the two collections are cleared and a cancel message $\varphi$ is returned where the `exclusions` set consists of the singleton $\pi$.`current` (meaning that further operations of the chosen `condition` should not be canceled). The complexity of the Selector is constant with an appropriate pseudo-random number generator for choosing a packet in lines 2 and 5.

*2.1.7 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 modeler 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 transformation modeler may handle these cases by modifying the transformation model. The complexity of the Synchronizer depends on the implementation of the custom merge function. Note that the complexity of the default merge function is linear in terms of the total number of graph nodes.

*2.1.8 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 a primitive is actively processing a packet $\pi$ such that the current condition of $\pi$ is not in $\varphi$.`exclusions`, where $\varphi$ 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`. Furthermore, resetting the dirty flag of modified nodes is done in the Composer by calling the `clean` method of a packet. Also, resetting the `success` and `fail` collections of the control primitives should be done by calling their `reset` method at the appropriate time.

## 2.2 Rationale

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 MTL, 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 be it rule-based, constraint-based, or function-based. In fact, some languages require units with only a pre-condition to satisfy, while others 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 pair 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 to 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 nonsense. 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. The Match allows one 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 (it is equivalent to passing parameters between rules). The MatchSet allows delaying 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 one to join branches or threads

of concurrent transformations. Also, having separated the matching from the rewriting enables one 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. If the Rollbacker is used in combination with the Iterator, then the overall behavior can handle back-tracking for cases where multiple matches are found.

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 one to modularly organize T-Core primitives. It serves as an interface to the primitives it encapsulates. The Composer is the extension point of T-Core where arbitrary transformation units can be defined by combining any of the presented primitives via specialization. This then enables scaling of transformations built on T-Core to large and complex model transformation 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 [31]. We believe however that such new constructs should either be part of the (programming or modeling) language or the tool in which T-Core is integrated, to keep T-Core as primitive as possible.

### 2.3 Usage of T-Core

The API of T-Core presented in the previous section offers a common interface to all primitive transformation operators. Furthermore, the CompositionPrimitive can be used to encapsulate the execution of other primitives in order to provide abstraction. The `packetIn` method is the entry point of a T-Core transformation. Fig. 2(a) illustrates a typical interaction with a transformation operator. When a CompositionPrimitive gets initially created, it is responsible of recursively created the instances of its sub-primitives following the composite design pattern [19]. Its `packetIn` is invoked with a packet that had previously been initialized with the input graph of the transformation. Because the operators support asynchronous execution, the `packetIn` method returns the resulting packet after being processed by the corresponding RulePrimitive $r$. To know whether $r$ has been successfully applied or not, one should query the `isSuccess` property of $r$. Similarly, if an exception occurred, the `exception` property of $r$ will refer to the corresponding detailed error. Therefore it is important to not forget to set the `isSuccess` property of a custom Composer in case of a successful execution so that the invoking context of the transformation can be aware of that status, as well as any exception that may have occurred. A similar pattern can be used for the `nextIn` method.

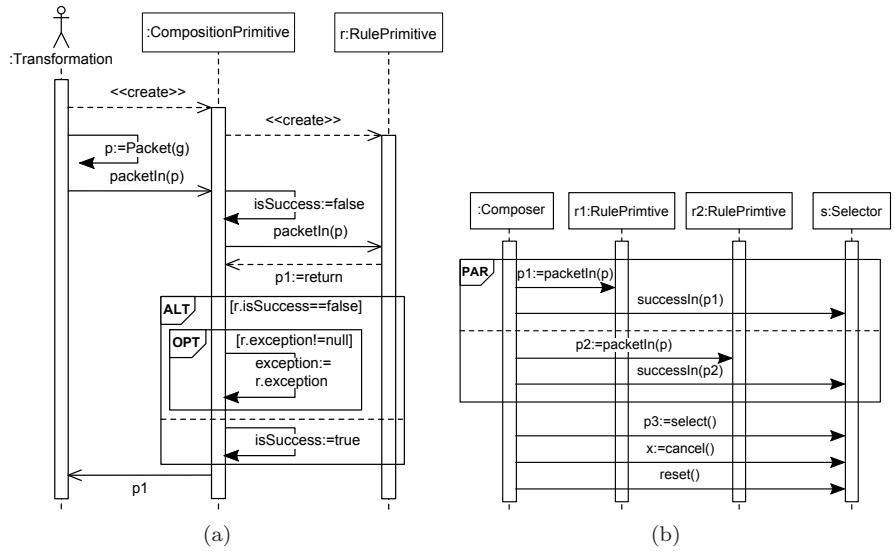



**Fig. 2** Sequence diagrams for using a RulePrimitive in (a) and a ControlPrimitive in (b).

Fig. 2(b) illustrates a typical interaction with a Selector. Recall that control primitives accumulate packets and can then produce a single packet: they serve as join points in the transformation. Packets are stored by invoking the `successIn` or `failIn` methods. At the appropriate time, the corresponding join function (`select` for the Selector and merge for the Synchronizer) can be invoked to retrieve a single packet from that operator. In the case of a Selector, a cancel event may be requested to invoke the cancelIn method of the other primitives if the suspension of their activity is desired. The `reset` method shall be invoked afterwards to clear the lists of packets. As in the previous case, verifying for success and errors needs to be integrated as well.

T-Core is designed in such a way that the transformation primitives can be executed independently from one another. To produce a meaningful result (transformation, query, state exploration, etc.), certain operators should preferably be applied before others. For example, Fig. 3 illustrates the interaction between T-Core operators to execute a transformation rule. In the following, we outline good practices for using the T-Core primitive operators:

- A Matcher should always be preceded by an Iterator in order to select a match found.
- A Rewriter should not be applied before executing the Matcher whose condition is the pre-condition of the Rewriter's post-condition. Otherwise, the rewriting phase will not affect the input graph and an exception will be reported.
- A Roll-backer will typically receive a `packetIn` message as soon as the enclosing Composer receives a packet. That way, the original graph will be checkpointed and may be restored at a later time via the `nextIn` method.

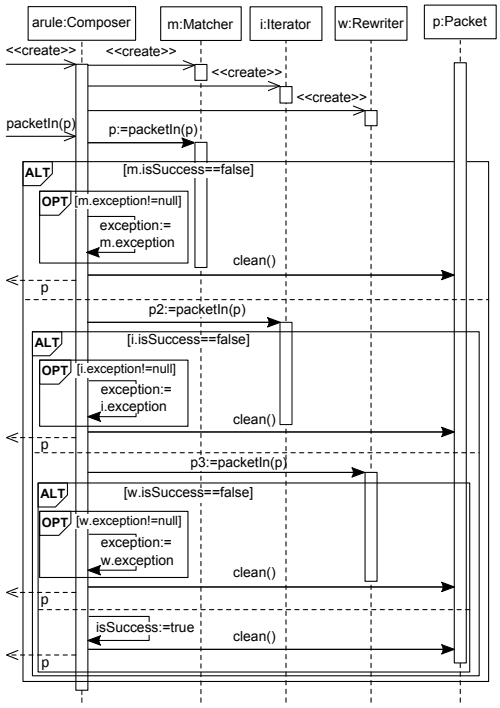



**Fig. 3** Sequence diagram of a simple rule execution.

– A Resolver should be executed after at least one application of a Rewriter. That is because conflicts are detected in an optimistic way, *i.e.,* after a modification of the graph, since the invariant part of the rule is not stored.

## 3 Implementation of T-Core

We implemented T-Core with a mindset to execute model transformations on large models efficiently. Therefore we provide an efficient implementation of the model transformation primitives. The details of the implementation of the data structures used to represent models to-be-transformed as well as transformation models are available at [54]. Models are encoded as graphs implemented with the IGraph library [11]. It also compares the efficiency of T-Core based transformations with other existing transformation tools. In this section we focus on implementation of the two core operators: the Matcher and the Rewriter.

Pattern matching and, in particular the sub-graph homomorphism problem, is NP-complete [38]. There are however various exponential-time worst case solutions for which the average-time complexity can be reduced with the help of heuristics. These approaches can be divided into two major categories: *search plans* and *constraint satisfaction problems* (CSP).

Search plan techniques [62, 20] define the traversal order for the nodes of the model to check whether the pattern can be matched. This is done by computing the cost tree of the different search paths and choosing the least costly one. Complex model-specific optimization steps can be carried out for generating efficient adaptive search plans [58]. Examples of such heuristics are the use of typing information with respect to meta-model elements or the use of cardinality constraints defined in the meta-model.

Graph pattern matching can also be described as a constraint satisfaction problem [45], where the pre-condition elements are variables, the elements of the model form the domain and typing, and the links and attribute values form the set of constraints. These techniques make use of backtracking algorithms [30] for finding a sub-graph of the input graph that is isomorphic[3] to the pre-condition graph. The algorithm explores the search space in a depth-first order. Well-known algorithms such as Ullmann [56] and VF2 [10] are some of the most efficient for solving the sub-graph isomorphism problem as a constraint satisfaction problem. Like in Ullmann's approach, VF2 constructs a search-tree traversing the host graph depth-first and backtracks when the current search-state fails a compatibility test. The algorithm also performs a pruning of the search space during the matching process. The major difference between Ullmann and VF2 is that, within one backtracking step, Ullmann compares pairs of adjacent nodes, while VF2 compares a node with its neighborhood. Moreover, Ullmann's approach verifies the semantic compatibility between pairs of nodes in the match, while VF2's feasibility test ensures a correct structure of the match. A combination of VF2 and Ullmann for hierarchical graphs was proposed in [43]. The idea was to merge the two search plans providing containment edges and local edges to denote hierarchy. The time complexity was thus improved.

Search-plan techniques are known to be more efficient than CSP algorithms [20]. However, most of these techniques [17, 6, 62] are based on LHS-RHS rule couple and do not allow to dissociate the matching part from the rewriting part. CSP implementations is that rules can be interpreted instead of compiled. The advantages of an interpreter is the execution of *true* in-place transformations and support for arbitrary nesting and amalgamation of the rules [2]. Furthermore in this context, self-modifying rules may alter their execution at run-time. Note that the contribution of this article is *not* about the performance of T-Core, but rather its expressiveness. For these reasons, we chose to implement the pattern matching of the Matcher as a CSP backtracking algorithm.

### 3.1 An Efficient Sub-graph Isomorphism Algorithm

The matching algorithm of the Matcher combines our own variation of the VF2 algorithm together with the refinement strategy of Ullmann's algorithm,

[3] In fact, it is homomorphic since the added attribute constraints in the pattern graphs describe constraints on the attributes of the source graph.

as outlined in Algorithm 10. The procedure `extend` augments the state of the

**Algorithm 10** `extend`(state)

```
1: if mappingIsComplete(state) then
2:    storeMatch(state)
3:    return
4: for p, s in suggestMapping(state) do
5:    if areCompatible(p, s) then
6:       if areSyntacticallyFeasible(p, s) then
7:          if areSemanticallyFeasible(p, s) then
8:             state.storeMapping(p, s)
9:             extend(state)
10:            state.undoMapping(p, s)
```

algorithm with all possible mappings from the pattern graph to the source graph. In the following, we call a *mapping* the one-to-one correspondence between a pattern node and a source node. We denote by a *match* the set of mappings in which all source nodes form a graph that is homomorphic to the pattern graph. Lines 4-14 recursively compute further mappings given the current state of the algorithm. The *state* stores the following information:

- $M^P$ and $M^S$ are the mapping sets holding the pattern nodes and the source nodes respectively in the current mappings,
- $T^P_{out}$ and $T^S_{out}$ hold the set of adjacent nodes to respectively $M^P$ and $M^S$ following outgoing edges, at any time;
- $T^P_{in}$ and $T^S_{in}$ hold the set of adjacent edges coming in respectively $M^P$ and $M^S$ following incoming edges, at any time;
- $T^P_{inout} = T^P_{out} \cap T^P_{in}$ and $T^S_{inout} = T^S_{out} \cap T^S_{in}$.

$T^P_{out}, T^P_{in}, T^P_{inout}$, and $T^S_{inout}$ are called the *terminal sets*. Each step of the search computes a partial mapping of the nodes and verifies that it does not violate the topology of the pattern graph. `suggestMapping` suggests a potential mapping of a source node $s$ with a pattern node $p$ (the pair $(p, s)$ is also known as the candidate pair in [10]). The choice of the pair is done in the following order: first from $(T^P_{inout}, T^S_{inout})$, then from $(T^P_{out}, T^S_{out})$, then from $(T^P_{in}, T^S_{in})$, and finally from all other nodes.

Afterwards, `areCompatible` verifies if it is worth continuing this mapping. This is done by comparing the number of incident edges of $s$ and $p$ (this is known as the refinement step in [56]). The compatibility check verifies that:

$$|Out(p)| \leq |Out(s)| \wedge |In(p)| \leq |In(s)| \tag{1}$$

where $In(n)$ and $Out(n)$ respectively represent the set of incoming and outgoing adjacent edges of a node $n$. This is similar to the refinement step of Ullmann's algorithm.

Then come the feasibility checks. `areSyntacticallyFeasible` ensures that the topology of the current mapping corresponds to a sub-graph of the pattern

graph. This is done by looking at the number of incident edges when $(p, s)$ is added to the current set of mappings ($M^P$ and $M^S$).

Let $InOut(n) = In(n) + Out(n)$, for any node $n$,
let $Out_p = Out(p) \cap T^P_{out}$ and $Out_s = Out(s) \cap T^S_{out}$,
let $In_p = In(p) \cap T^P_{in}$ and $In_s = In(s) \cap T^S_{in}$,
let $All_p = M^P \cup T^P_{out} \cup T^P_{in}$ and $All_s = M^S \cup T^S_{out} \cup T^S_{in}$.

Then the following must be true to ensure syntactic feasibility of $s$ and $p$:

$$|Out_p| \leq |Out_s| \wedge |In_p| \leq |In_s| \quad \wedge \tag{2}$$
$$|Out_p| + |In_p| + |InOut(p) - All_p| \leq |Out_s| + |In_s| + |InOut(s) - All_s|$$

The last test ensures that the semantics of $s$ corresponds to the semantics of $p$. In our case, semantic information of the nodes is encoded in their attributes, but the details of the function `areSemanticallyFeasible` will be elaborated later on. When $s$ and $p$ satisfy all of the above conditions, $(p, s)$ is considered a valid mapping and is stored in the state (line 8). The algorithm then continues looking for remaining mappings. When all valid mappings have been computed (lines 1-3), the corresponding match is stored. The algorithm backtracks to the previous state when either a complete match is found or if the current partial match (set of mappings in $M^P$ and $M^S$) does not allow for any further valid mapping. Note that a nice property of this algorithm is that any state in the search tree is visited exactly once.

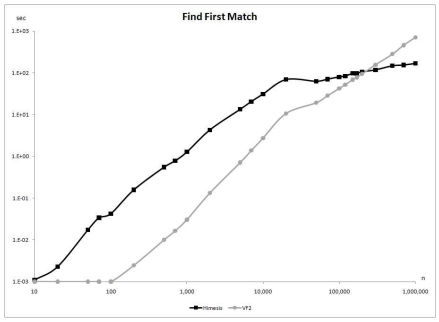
Algorithm 11 allows us to compute all matches between a pattern graph $P$ and a source graph $S$. Furthermore, an initial set of mappings can be specified to prune the search tree constructed by the procedure `extend`. This initial mapping can also be seen as the initial context in which the matchings must be computed: it restricts specific pattern nodes to be mapped exactly to predefined source nodes.

**Algorithm 11** `computeMappings`(S, P, context)

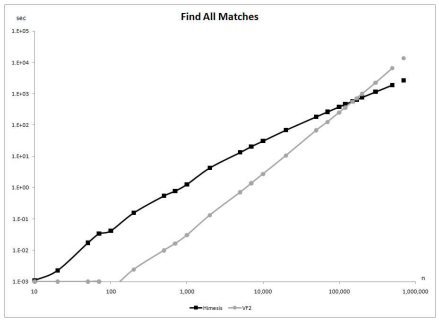
```
1: state ← initState(S, P)
2: for p, s in context do
3:     state.update(p, s)
4: extend(state)
5: return state.getMatches()
```

### 3.2 Performance Evaluation of the Implementation

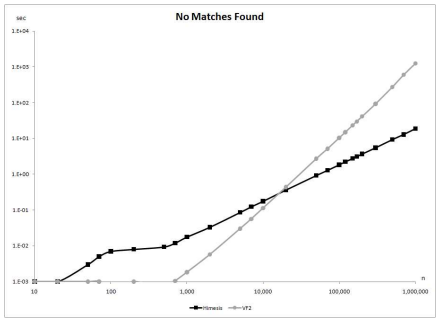
Let us first analyze the space complexity of the `extend` procedure. The state of the algorithm is encoded in the *state* variable. It holds the two partial mapping sets as well as the all terminal sets. Thus, the number of nodes stored in the state is at most $5 \times |V(P)| + 3 \times |V(S)|$ which is linear in terms of the nodes of the source and pattern graphs. Moreover, since IGraph stores the nodes as integers, *state* is quite compact. Additionally, the experiments below show




(a)




(b)




(c)

| Nodes | | Edges | Class | Attribute | Associations | Matches | All | First | None |
|---|---|---|---|---|---|---|---|---|---|
| Small | $[0,10^3[$ | 399 | 51 | 45 | 51 | 729 | 0.016 | 0.04 | 0.034 |
| Medium | $[10^3,10^5[$ | 33,727 | 4,133 | 4,249 | 4,196 | 83,766 | 0.067 | 0.125 | 0.348 |
| Large | $[10^5,10^6]$ | 304,552 | 38,158 | 38,312 | 37,919 | 891,171 | 2.929 | 5.006 | 22.305 |

(d)

**Fig. 4** Size average of sub-graph isomorphism matching over the six pattern graphs. The graphs are plotted on a log-log scale.

that the algorithm performs better if the adjacency list (encoded as a hash table) is memorized as well. The size of this hash table is in the worst case $|V(P)|^2 + |V(S)|^2$ for fully connected, directed, simple graphs.

We now compare the time performance of the `extend` algorithm of T-Core's Matcher with VF2's sub-graph isomorphism algorithm. We have chosen the IGraph implementation of VF2 as a benchmark which is in direct correspondence with the original implementation. Note that T-Core is implemented in Python whereas VF2 was implemented in C. According to [18], Python is in general slower than C by an average factor of 23, which is not integrated in the results presented here. In these experiments, we gathered the computation time with respect to the number of nodes $n$ of the source graph. The source graph represents random valid class diagrams. The average number of class diagram elements is shown in Fig. 4(d). For each source graph we have run the algorithm on six pattern graphs whose sizes range from 2 to 12 nodes. Our experience shows that this is a typical size for LHS and NAC pre-condition patterns assuming an expressive control flow language such as MoTif [55]. For both the source and pattern graphs, the number of edges is the same order as the number of nodes (which is typical in class diagrams). Each data point of the plots in Fig. 4 represents the average time over the six pattern graphs.

Fig. 4(a) shows the performance of both algorithms for finding the *first* match only. For small graphs, VF2 is about 25 times faster than T-Core. For medium graphs, VF2 is twice as fast as T-Core. However, at around $2.2 \times 10^5$ nodes, both perform equally fast. At this point, T-Core overtakes VF2 by a factor of 6 for large graphs.

Fig. 4(b) shows the performance of both algorithms for finding *all* matches. For small graphs, VF2 is about 60 times faster than T-Core. For medium graphs, VF2 is 5 times faster than T-Core. However at around $1.5 \times 10^5$ nodes, both perform as fast. At this point, T-Core overtakes VF2 by a factor of 5 for large graphs.

Fig. 4(c) shows the performance of both algorithms when *no match* exists. For small graphs, VF2 is about 24 times faster than T-Core. The medium graph category must be divided into two. For graphs with $10^3$ to $10^4$ nodes, VF2 is 3.6 times faster than T-Core. As for graphs with $10^4$ to $10^5$ nodes, T-Core overtakes by a factor of 2.2. The break even point is around $1.7 \times 10^4$ nodes. At this point, T-Core overtakes VF2 by 3 times for large graphs.

The table in Fig. 4(d) summarizes these observations. Notice how T-Core significantly outperforms VF2 for large graphs.

### 3.3 Pattern Matching

The transformation kernel of the new version of AToM$^3$ [34] is T-Core. In T-Core, the pre- and post-condition patterns of a rule are encoded as graphs in IGraph. A pre-condition is composed of a positive condition graph (LHS) and optional negative condition graphs (NACs). The semantics is as follows: if an occurrence of the LHS is found in the source graph before the rule is applied and none of the NACs are found, then an occurrence of the RHS must be found in the source graph after the rule has been applied. A more formal definition based on category theory can be found in [15].

In our implementation, a node $n$ of a pattern graph holds the following information:

- A universally unique identifier: such identifiers are ensured to be unique at all time.
- The type $t$ of the model element $n$ encodes: the absolute path (across packages) of the name of the type element.
- A boolean flag *stm* specifying whether a source node mapped to $n$ must be of type $t$ or a sub-type of $t$.
- The set *st* of all sub-types of $t$.
- The identifier of a binding pivot $\overleftarrow{x}$ (for pre-condition graphs). If specified, it predefines which source node that was assigned to the pivot $x$ must be matched to $n$.
- The identifier of a pivot assignment $\overrightarrow{x}$. If specified, it indicates that the source node mapped to $n$ will be assigned to the pivot $x$.
- A label global to the scope of the rule. Node labeling in the different pattern graphs of the rule is used as follows. In the LHS, a label allows one to

distinguish between two nodes of the same type that must be mapped to different source nodes. A label present in both the LHS and the RHS or in both the LHS and a NAC corresponds to the same matched source node. A label present in a NAC but not in the LHS allows one to distinguish between two nodes of the same type that must be mapped to different source nodes.
- Each attribute of the meta-model element corresponding to $t$ is subject to the RAM procedure [32]. In the LHS and the NAC, the node is assigned one constraint per attribute. The constraint can be of arbitrary complexity, but can only refer to source nodes bound to the corresponding pattern (LHS xor NAC). In the RHS, the node is assigned an action code per attribute. The action can be of arbitrary complexity, but can only refer to source nodes bound to the LHS pattern.

The size of the data stored in each pattern node is 1,342 bytes, without taking into consideration the meta-model attributes. Additional information is stored at the graph pattern level: the set of all meta-models involved in the pattern[4] as well as an additional constraint (for a LHS or a NAC) or action (for an RHS). The constraints and actions are treated similarly to pattern node attributes.

Up to now, we have described an efficient solution for finding a sub-graph of the source graph isomorphic to the pattern graph. However, this is not sufficient for pattern matching as it only takes into account the topology of the pattern graph. Constraints attached to match patterns as well as NACs must be taken into consideration as well. Algorithm 12 specifies a procedure that modifies the previous sub-graph isomorphism solution for pattern matching purposes. We must first modify the `extend` procedure to handle constraints on meta-model attributes and node typing. The type of a pattern node $p$ and a source node $s$ must correspond. This requirement must be verified as early as possible to reduce the search space. We therefore modify the function `areCompatible` in Algorithm 10. More specifically, condition (1) must now take into consideration the types of the candidate pair $(p, s)$ as specified in (3), such that the type of $s$ is the same as the type of $p$ or one of its sub-types. (1) can then be rewritten as:

$$|Out(p)| \leq |Out(s)| \wedge |In(p)| \leq |In(s)| \wedge ((s.t = p.t) \vee (p.stm \wedge s.t \in p.st)) \quad (3)$$

Additionally, the function `areSemanticallyFeasible` must ensure that the attributes held in $s$ each satisfy the corresponding meta-model attribute constraints in $p$. Also, to help the algorithm find a match as soon as possible, we have parametrized the `suggestMapping` function with a priority mechanism to suggest a candidate pair. Our implementation allows us to specify an arbitrary order of a terminal set. By default, `suggestMapping` will suggest an unmatched pattern node such that its type occurs the least often in the graph. This heuristic ordering can be modularly extended with further knowledge of the pattern graph and the source graph.

[4] Because in AToM$^3$, rules can involve many meta-models as in *e.g.,* multi graph grammars [29].

**Algorithm 12** `match`(G, LHS, context)

```
 1: validMatches ← ∅
 2: moreNACs ← False
 3: for NAC in LHS.getNACs() do
 4:     bridge ← NAC.getBridge()
 5:     if V(NAC.getBridge()) > 0 then
 6:         moreNACs ← True
 7:     else
 8:         for nacMatch in computeMappings(G, NAC, context) do
 9:             if NAC.checkConstraint(nacMatch) then
10:                 return ∅
11: if not moreNACs then
12:     for lhsMatch in computeMappings(G, LHS, context) do
13:         if LHS.checkConstraint(lhsMatch) then
14:             validMatches ← validMatches ∪ {lhsMatch}
15:     return validMatches
16: maxNAC ← LHS.getNACwithMaxBridge()
17: B ← maxNAC.getBridge()
18: for bMatch in computeMappings(G, B, context) do
19:     for maxNACMatching in computeMappings(G, maxNAC, bMatch ∪ context) do
20:         if not maxNAC.checkConstraint(maxNACMatching) then
21:             goto 20
22:     for lhsMatch in computeMappings(G, LHS, bMatch ∪ context) do
23:         if LHS.checkConstraint(lhsMatch) then
24:             for NAC in LHS.getNACs() do
25:                 if NAC ≠ maxNAC and V(NAC.getBridge()) > 0 then
26:                     for nacMatch in computeMappings(G, NAC, lhsMatch ∪ context) do
27:                         if not NAC.checkConstraint(nacMatch) then
28:                             validMatches ← validMatches ∪ {lhsMatch}
29: return validMatches
```

The pattern matching algorithm of the Matcher is described in Algorithm 12. The procedure `match` takes a source graph $G$ and the LHS pattern graph as input. Pivot bindings may also be specified in the *context*. The procedure can be one of three cases. In the following, we consider a match as *valid* if the source nodes in the mappings of the match satisfy the constraint of the pattern graph.

**No NACs.** When there are no NACs specified in the pre-condition pattern, then only lines 1-3 and 11-15 are applied. This simply calls the `computeMappings` procedure and returns the valid matches.

**Unbound NACs.** We denote a NAC as unbound if none of its nodes has a label present in the corresponding LHS. If the pre-condition has unbound NACs, it suffices to find one valid NAC match to prevent the pre-condition pattern from successfully finding any matches. Lines 3-10 describe this behavior. First, $G$ is matched on the NAC with the provided context. If no valid match is found, the procedure then tries to find matches for the LHS as in the previous case. Otherwise, no match is output.

**Bound NACs.** All other NACs are bound to the LHS (lines 16-29). Since `computeMappings` is the most costly procedure, we want to avoid computing mappings twice, *i.e.,* the common part between the LHS and a NAC. Thus

the idea is to first match the common part between the LHS and a NAC, then continue the matching along the NAC, and finally, if no valid NAC matches were found, continue from the match of the common part along the LHS.
A NAC having a common part with the LHS means that there is a sub-graph of the LHS that overlaps with the NAC. We denote this intersection as a pre-condition graph called *bridge*. In general, computing the bridge would require us to find the maximum common sub-graph (MCS) between these two graphs. Solving the MCS isomorphism problem is NP-Complete. However, making use of the labels in the pattern graphs reduces the complexity to linear-time. Therefore the bridge can be constructed as follows: if a node has a label present in nodes of both the LHS and the NAC, then this node is part of the bridge. Also, every edge in the smallest graph between the LHS and the NAC whose source and target nodes are in the bridge is part of the bridge. However, recall that pattern nodes also hold a constraint for each meta-model attribute. Thus, each meta-model attribute of a bridge node is computed as the conjunction of the corresponding attribute constraint in the LHS and the corresponding attribute constraint in the NAC. Note that no constraint is added on the pattern graph of the bridge as in the LHS or NAC cases. It is easy to show that the time complexity of constructing the bridge between the LHS and an NAC is $O(V+E)$, where $V = max(|V(LHS)|, |V(NAC)|)$ and $E = min(|E(LHS)|, |E(NAC)|)$[5].
In the `match` procedure, line 17 computes the bridge $B$ with the largest number of nodes. Since a bridge can be statically computed, all bridges have already been precomputed and integrated in the corresponding NACs (at compile-time). On line 18, $G$ is matched on $B$ with the provided context. Then on lines 19-21, $G$ is matched on the NAC corresponding to $B$. To prune the search space of this matching, the bridge mappings are provided as context together with the initial context. Those mappings are valid since the nodes in $B$ are in the NAC as well. If a valid match for this NAC is found, then the current match of $B$ is discarded and the next one is tried. When a match of $B$ is found such that it does not induce a valid match, we match $G$ on the LHS with again the bridge mappings provided as context together with the initial context. Each valid match of the LHS represents a potential valid match of the procedure. However, there may be additional bound NACs with a bridge having less nodes than $B$. In this case, lines 24-28 ensure that only the valid matches of the LHS that do not satisfy the remaining NACs are stored. Note that when applying the `computeMappings` procedure on $G$ with the remaining NACs, the LHS mappings are provided as context together with any pivot node bound in the LHS that were given in the initial context. Finally on line 29, only the valid matches are output.

When multiple matches need to be found, it would be overly expensive to execute the whole procedure every time. For this reason, our implementation

[5] $V$ should also be multiplied by the maximum number of meta-model attributes, which is small in practice.

relies on the *iterative pattern* [19] which enables the procedure to halt after a complete match is found and the search is resumed on the next invocation.

### 3.4 Rewriting the Matches

A rule is successfully applied when its pre-condition is satisfied. The pre-condition satisfaction is ensured by the pattern matching algorithm described previously. One way to satisfy the post-condition is to modify the matched nodes in the source graph appropriately. To transform (or rewrite) the matches, a RHS pattern graph is provided with a compiled `execute` function encoding the appropriate modification actions, which is invoked by the Rewriter. Given the LHS and the RHS pattern graphs, the rewriting of a match $M = \{(p, s)|p \in \text{LHS} \wedge s \in G\}$ can be statically determined. For each $(p, s) \in M$ we perform the following steps in order:

1. If the label of $p$ is present in both the LHS and the RHS, then an *update operation* is executed. Each attribute of $s$ is set according to the action specified in the corresponding meta-model attribute of the RHS node that has the same label as $p$.
2. Let $C$ represent the graph whose node labels are present in the RHS but not in the interface graph $K$. Also edges of $C$ are constructed in a similar way as for the bridge, *i.e.,* $E(C) = \{(n_i, n_j)|n_i, n_j \in V(C) \wedge (n_i, n_j) \in E(\text{RHS})\}$. Then a *create operation* is applied to the nodes and edges of $C$. For each node (or edge) in $V(C)$ (or $E(C)$), a corresponding source node (edge) is created in the source graph. Furthermore, the attributes of the new nodes are initialized according to the action specified in the corresponding meta-model attribute of the respective node in $C$.
3. If the label of $p$ is present in the LHS but not in the RHS, then a *delete operation* is applied and removes $s$ from the source graph. Note that in IGraph, deleting a node automatically deletes its adjacent edges.
4. If $p$ is assigned a pivot identifier $\vec{x}$, then $\vec{x}$ will be mapped to $s$.
5. Finally, after all nodes have been processed, we apply the action specified in the RHS on the source nodes that are in $M$ as well as those created from $C$.

The run-time complexity of the Rewriter is linear: $O(|V(\text{LHS})| + |E(\text{LHS})| + |V(\text{RHS})| + |E(\text{RHS})|)$. Note that according to graph transformation literature [14], T-Core's transformation procedure follows the Single-Pushout (SPO) approach in contrast with the Double-Pushout (DPO) approach. On the one hand, the identification issue of the gluing condition in DPO is avoided thanks to the labeling mechanism in place. That is because every node in each pattern graph is unique and thus may be mapped to exactly one node in each matching. On the other hand, we have explicitly chosen to solve the dangling edges issue automatically. That is if a matched source node must be deleted, all its adjacent edges will be deleted too. This has the advantage of reducing the number of rules in the transformation.

For an in-depth analysis of the performance of T-Core as well as a performance comparison with other MTLs, we refer the reader to the following technical report [54].

## 4 Transformation Language Product Line

T-Core empowers the transformation developer to build transformation units that are specific to the problem the transformation will solve. General-purpose transformation languages, such as QVT or ATL, encumber the developer with unneeded features. This adds complexity to the design the transformation which may lead to design errors and even reduce productivity [27]. In contrast, MTLs based on T-Core enable designing transformation models that are tailored to the problem domain. This helps obtaining more optimal and simpler solutions than with general-purpose transformation languages.

There is a wide variety of transformation languages and tools that exist today. Also, they are very powerful in solving the problems they were initially intended for. For example, FUJABA [41] is primarily meant to provide reverse-engineering capability, AToM$^3$ [34] and GReAT for defining translational semantics and simulation of formalisms, the new version of VIATRA [7] to provide means for model synchronization, etc. However, most of them have a tendency to provide a generic tool for solving any kind of model transformation problem. This is especially true with the arrival of QVT and most applications of ATL [42]. This genericity requires transformation languages to be very expressive, which makes analysis of transformation models built using these *general purpose* transformation languages very hard. In fact, some approaches have realized this problem and propose Turing-incomplete transformation languages, such as DSLTrans [5].

The solution proposed here is to use a sub-set of T-Core primitives to restrict a transformation language for one specific purpose or intention. To some extent, one can redefine a transformation language as consisting of the following features:

1. **Primitive transformation operators**, for example taken from (a subset of) the T-Core module;
2. Combined with a **scheduling language**, which can be programmed (*e.g.,* Java [16]) or modeled (*e.g.,* UML Activity diagrams [36], Colored Petri nets [61]).

In fact, the scheduling language may be a domain-specific language dedicated for defining transformation schedulers. The combination of both provides a product line of **problem-specific** transformation languages. This restricts the transformation engineer to focus entirely on designing transformation models without added complexity that is irrelevant for the purpose of the transformation. Also, the transformation language has no more expressiveness than is needed and this may allow for better analysis of the transformation models. Nevertheless, the expressiveness of the transformation language then depends

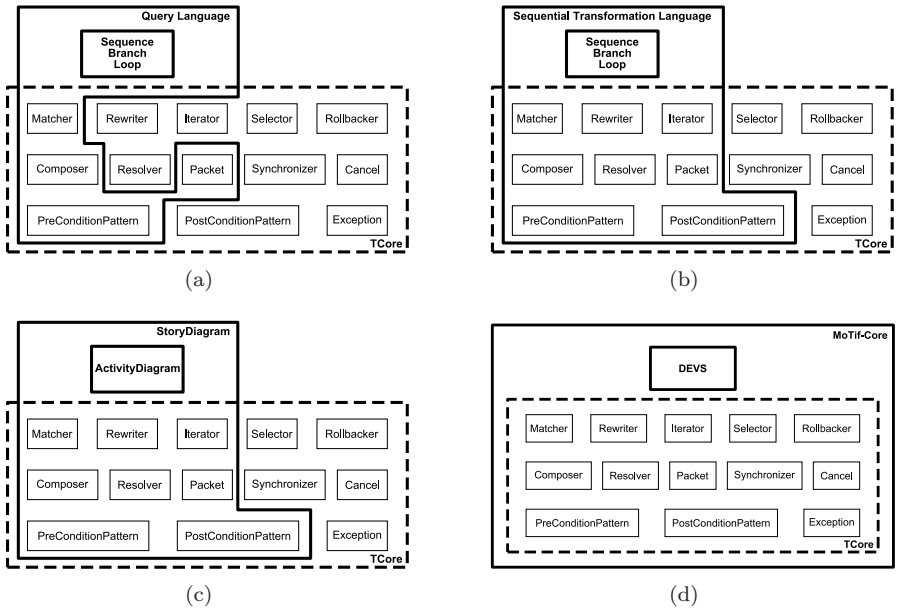



**Fig. 5** Combining T-Core with other languages allows one to re-construct existing and new languages.

on the glue language (*i.e.,* the scheduler) used and the primitive operators chosen.

De-constructing MTLs 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 modeling language allows us to re-construct some already existing transformation languages and even construct new ones. Fig. 5 shows some examples of combinations of T-Core with other languages. Fig. 5(a) and Fig. 5(b) combine a subset of T-Core with a simple (programming) language which offers *sequencing*, *branching*, and *looping* mechanisms (as proposed in Böhm-Jacopini's *structured program theorem* [8]). We will refer to such a language as an *SBL language*. The first combination only involves the Matcher and its PreCondition-Pattern, 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 (subgraph) 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. Fig. 5(b) enumerates the T-Core primitives combined with an SBL language necessary to design a complete sequential MTL. Replacing the SBL language by another one, such as UML Activity Diagrams in Fig. 5(c), allows us to re-construct Story Diagrams [17], for example, since they are defined as a combination of UML Activity and Collaboration Dia-

grams with graph transformation features. Other combinations involving the whole T-Core module may lead to novel transformation languages with exception handling and the notion of timed model transformations when combined with a discrete-event modeling language [53].

We now present the re-construction of two transformation features using the combination of an SBL language with T-Core as in Fig. 5(b). Then we present the construction of a new MTL by combining T-Core with a programing language.

### 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, an extension of UML Activity Diagrams. A Story Diagram organizes the behavior 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. Fig. 6(a) shows such a story diagram taken from the `doDemo` method example in [17]. 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 [22] 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 labeled `each 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. When all iterations have been completed, the flow proceeds with statement 8 reached by the transition labeled `end`, which simulates the elevator going one level up.

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. Fig. 6(b) describes the three necessary rules corresponding to the three Story Diagram activities. We use the visual concrete syntax of MoTif 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 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 pivot elements. A right-directed arrow on top of the label depicts that the model element matched for this pattern element is assigned to a pivot (*e.g.,* `p4` and `l4`). A left-directed arrow on top of the label depicts that the model element matched for this pattern element is bound to the specified pivot (*e.g.,* `this` and `e1`).

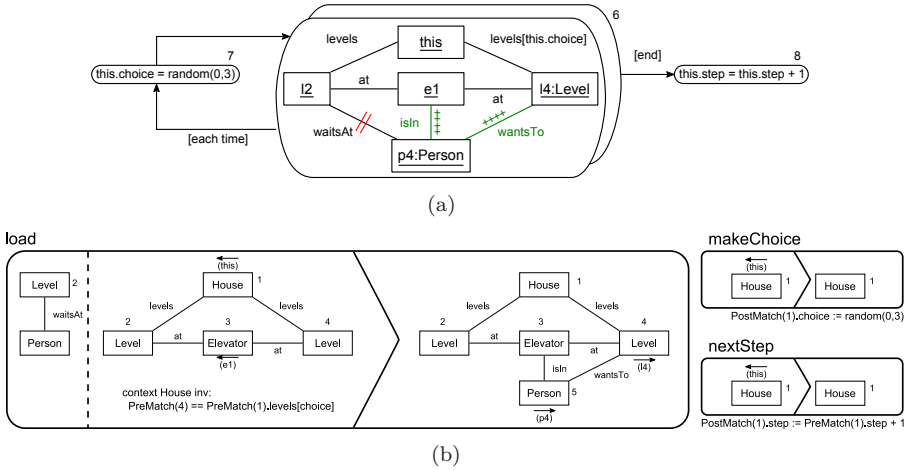



**Fig. 6** (a) The FUJABA doSubDemo transformation showing a for-all Pattern and two statement activities. (b) The three MoTif rules for the doSubDemo transformation.

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`. Algorithm 13 describes this behavior.

**Algorithm 13** doSubDemoC.`packetIn`($\pi$)

```
π ← loadC.packetIn(π)
π ← nextStepC.packetIn(π)
isSuccess ← true
return π
```

makeChoiceC and nextStepC behave as simple transformation rules. Their `packetIn` method behaves as specified in Algorithm 14. 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 the matcher 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 the rewriter 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 behavior

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 15 specifies that behavior. 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.

Having seen the overall T-Core transformation model, let us examine how the different Resolvers should behave in order to provide a correct and complete transformation. The first rewriter called is loadW and the first time it receives a packet is when a transformation is applied on one of the matches of the matcher 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 the resolver 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.

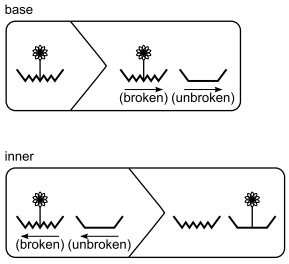



**Fig. 7** The transformation rules for the *Repotting Geraniums* example

**Algorithm 16** baseC.packetIn($\pi$)

```
isSuccess ← false
π ← baseM.packetIn(π)
if not baseM.isSuccess then
   return π
while true do
   π ← baseI.packetIn(π)
   if baseI.isSuccess then
      π ← baseW.packetIn(π)
      if not baseW.isSuccess then
         return π
      π ← baseR.packetIn(π)
      if not baseR.isSuccess then
         return π
      π ← innerC.packetIn(π)
   π ← baseM.packetIn(π)
   if not baseM.isSuccess then
      isSuccess ← true
      return π
```

**Algorithm 14** makeChoiceC.packetIn($\pi$)

```
isSuccess ← false
π ← makeChoiceM.packetIn(π)
if not makeChoiceM.isSuccess then
   return π
π ← makeChoiceI.packetIn(π)
if not makeChoiceI.isSuccess then
   return π
π ← makeChoiceW.packetIn(π)
if not makeChoiceW.isSuccess then
   return π
π ← makeChoiceR.packetIn(π)
if not makeChoiceR.isSuccess then
   return π
isSuccess ← true
return π
```

**Algorithm 15** loadC.packetIn($\pi$)

```
isSuccess ← false
π ← loadM.packetIn(π)
if not loadM.isSuccess then
   return π
π ← loadI.packetIn(π)
if not loadI.isSuccess then
   return π
while true do
   π ← loadW.packetIn(π)
   if not loadW.isSuccess then
      return π
   π ← loadR.packetIn(π)
   if not loadR.isSuccess then
      return π
   π ← makeChoiceC.packetIn(π)
   π ← loadI.nextIn(π)
   if not loadI.isSuccess then
      isSuccess ← true
      return π
```

## 4.2 Re-constructing amalgamated rules

Rensink *et al.* claim that the *Repotting the Geraniums* example is inexpressible in most transformation formalisms [44]. 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*. Fig. 7 illustrates the two nested graph transformation rules involved and Algorithm 16 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 behavior as loadC.`packetIn` in Algorithm 15, 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 returning the final packet in success mode.

### 4.3 Re-constructing non-deterministic rule selection

GReAT is a well-known graph transformation language with asynchronous behavior [1]. For example, Fig. 8 presents a Test block where two Cases (atomic or composite rules) can be applied. When a Test block receives a packet in GReAT, the packet is tested on all the Cases. If multiple Cases succeed, only one will be applied non-deterministically. Algorithm 17 shows how the Test block is re-constructed using T-Core primitives. The packets output from the Matcher of each rule are accumulated in a Selector S. The `select` method is responsible for the non-deterministic selection of a rule as previously described in Section 2.1.6. Then, the rule corresponding to the chosen packet is applied. The method `getRuleFromPacket` finds the rule whose Matcher corresponds to the condition of the packet. Note that no resolver is needed for the Test composer since at most one rule is executed.

### 4.4 Py-T-Core

Currently we have implemented T-Core in Python and it is available at the website [48]. It is a direct implementation of the class diagram of Fig. 1. Therefore, the combination of T-Core primitives with Python as a scheduling

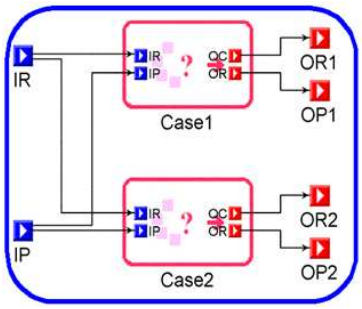



**Fig. 8** A Test block in GReAT showing two cases.

**Algorithm 17** TestC.packetIn($\pi$)

```
isSuccess ← false
for all r ∈ rules do
    π ← r.M.packetIn(π)
    if r.M.isSuccess then
        S.successIn(π)
    else
        S.failIn(π)
π ← S.select()
S.reset()
if not S.isSuccess then
    return π
r ← getRuleFromPacket(rules, π)
π ← r.I.packetIn(π)
if not r.I.isSuccess then
    return π
π ← r.W.packetIn(π)
if not r.W.isSuccess then
    return π
isSuccess ← true
return π
```

language seems adequate. This results in a new transformation language, called Py-T-Core[6]. It encapsulates re-usable idioms often found in existing MTLs.

For example, a query is defined as in Listing 1: given a packet, if a match is found it is selected and the resulting packet is output. The packet then consists of a single match set containing a single match. This match describes the sub-graph that satisfies the pre-condition pattern *i.e.,* the query.

**Listing 1** A query in Py-T-Core.

```
class Query(Composer):
  def __init__(self, LHS):
    super(Query, self).__init__()
    self.M = Matcher(condition=LHS, max=1)  # Find 1 match
    self.I = Iterator(max_iterations=1)  # Select the only match
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet)
    if not self.I.is_success: return packet
    self.is_success = True
    return packet
```

Listing 2 illustrates how a simple rule is defined following Algorithm 14.

**Listing 2** A simple rule in Py-T-Core.

[6] Similarly, an implementation in C would be called C-T-Core or in Java would be called J-T-Core.

```
class ARule(Composer):
  def __init__(self, LHS, RHS):
    super(ARule, self).__init__()
    self.M = Matcher(condition=LHS, max=1)  # Find 1 match
    self.I = Iterator(max_iterations=1)  # Select the only match
    self.W = Rewriter(condition=RHS)
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet)
    if not self.I.is_success: return packet
    packet = self.W.packet_in(packet)
    if not self.W.is_success: return packet
    self.is_success = True
    return packet
```

Listing 3 defines a transformation unit that applies a rule on all matches found.

**Listing 3** A rule applied on all matches at once in Py-T-Core.

```
class FRule(ARule):
  def __init__(self, LHS, RHS, max_iterations,
               external_matches_only, custom_resolution):
    super(FRule, self).__init__(LHS, RHS)
    self.M.max = max_iterations
    self.I.max_iterations = max_iterations
    self.R = Resolver(external_matches_only, custom_resolution)
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)  # Find all matches
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet)
    if not self.I.is_success: return packet
    while True:
      packet = self.W.packet_in(packet)
      if not self.W.is_success: return packet
      packet = self.R.packet_in(packet)
      if not self.R.is_success:  # Resolve any conflicts if needed
        self.exception = self.R.exception
        return packet
      packet = self.I.next_in(packet)  # Choose another match
      if not self.I.is_success:  # No more iterations are left
        self.is_success = True
        return packet
```

Listing 4 defines a transformation unit that applies a rule as long as there are matches. This is similar to what was described in Algorithm 16, with the difference that the latter also had a nested rule applied inside the loop.

**Listing 4** A rule applied as long as possible in Py-T-Core.

```
class SRule(ARule):
  def __init__(self, LHS, RHS, max_iterations):
    super(SRule, self).__init__(LHS, RHS)
```

```
    self.I.max_iterations = max_iterations
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet)  # Choose the 1st match
    if not self.I.is_success: return packet
    while True:
      packet = self.W.packet_in(packet)
      if not self.W.is_success: return packet
      self.is_success = True  # Rule was successfully applied once
      if self.I.iterations == self.I.max_iterations: return packet
      packet = self.M.packet_in(packet)
      if not self.M.is_success: return packet
      packet = self.I.next_in(packet)
      if not self.I.is_success: # No more iterations are left
        return packet
```

Py-T-Core allows a programmed[7] software to integrate with model transformation solutions thanks to the T-Core API. This is a pragmatic solution to bridge the gap between software developers who program large-scale systems and domain experts who describe the behaviors of their model through transformation. Other solutions to this problem exist, such as EMFTiger [16] where the integration is restricted to Java.

## 5 Case study: Re-engineering DSLTrans with T-Core

T-Core has been designed to represent the basic building blocks of transformation languages. To demonstrate its power to handle different types of transformation languages and provide empirical evidence that it can do so, requires choosing representative transformation languages and confirming they can be re-engineered to work through T-Core.

DSLTrans [5] is one such transformation language. DSLTrans is a Turing incomplete language that guarantees the confluence of any transformations designed using it. It also guarantees any transformation will terminate. A T-Core implementation will automatically benefit from these properties of DSLTrans. DSLTrans is built on a series of object and connection elements (shown in Table 2) to handle the transformation to the output model. It provides several interesting constructs that need a T-Core implementation to provide a straightforward mapping from a given DSLTrans transformation to a T-Core transformation. We have developed a T-Core plug-in for AToM$^3$. Models and rules can be translated from AToM$^3$ to T-Core graphs and compiled rules. Certain restrictions on the final DSLTrans output model also require special T-Core implementation. An appropriate Py-T-Core transformation then schedules the application of the rules.

Several challenges are presented when translating DSLTrans to T-Core. Since DSLTrans is built to transform Ecore-based models [21], DSLTrans re-

[7] As opposed to a modeled software where no artifacts are hard-coded.

| Objects | | Connections |
|---|---|---|
| | | ApplyAssociation |
| AnyMatchClass | FilePort | AttributeRef |
| ApplyAttribute | Layer | ExplicitSource |
| ApplyClass | MatchAttribute | Import |
| ApplyModel | MatchModel | NegativeIndirectAssociation |
| Atom | MetaModelIdentifier | NegativeMatchAssociation |
| AttributeRef | NegativeMatchClass | PositiveBackwardRestriction |
| Concat | Rule | PositiveIndirectAssociation |
| ExistsMatchClass | WildCard | PositiveMatchAssociation |
| | | PreviousSource |

**Table 2** DSLTrans elements.

spects the standard EMF constraint that every meta-model has a root element type that all other elements are contained within directly or indirectly. DSLTrans is also outplace: it builds its output model separately from the input model it is transforming and there can be no elements from the input model in the output model at any time. DSLTrans does maintain traceability links between the new elements and the associated original elements during transformation, but that is not reflected in the final output model. The Py-T-Core implementation must handle these restrictions. For specific rule scheduling, DSLTrans uses a sequence of layers. Other DSLTrans elements have different uses depending on which other transformation elements are connected to them. Finally, DSLTrans has a concept of indirect containment, where a rule precondition succeeding can depend on whether two elements are connected by recursive association. The Py-T-Core implementation must provide procedures and approaches for handling these scenarios as well.

In the sequel, we describe the main challenges in the DSLTrans re-construction process, focusing on the most complex constructs. For the full description of the translation, the reader can refer to the experience report [35].

### 5.1 Layers

A DSLTrans transformation consists of layers. Rules within layers are executed in a non-deterministic order until none of them can applied again. Applying a DSLTrans Rule does not make it ineligible to be re-applied. The corresponding Py-T-Core transformation unit is the BSRule shown in Listing 5. It consists of looping over a BRule (c.f. Algorithm ) that applies rules/branches until we run out of successful branches. Each DSLTrans Layer will have an analogous Py-T-Core BSRule.

**Listing 5** Selects a branch in which the matcher succeeds, as long as matches can be found.

```
class BSRule(Composer):
  def __init__(self, branches, max_iterations=INFINITY):
    super(BSRule, self).__init__()
    self.brule = BRule(branches)
```

```
    self.max_iterations = max_iterations
    self.iterations = 0
  def packet_in(self, packet):
    self.is_success = False
      while self.iterations < self.max_iterations:
        packet = self.brule.packet_in(packet)
        if not self.brule.is_success: return packet
        else: self.is_success = True
        self.iterations += 1
        return packet
```

Each layer, other than the first, is connected to the previous layer. In Py-T-Core, we define a Sequence to sequentially run each BSRule. The primary scheduler will pass the collection of layers to the Sequence in the correct order.

## 5.2 Positive Indirect Associations

DSLTrans supports deep containment relationships. Since this is not directly supported in T-Core, it had to be broken down into several steps to be implemented in Py-T-Core. This requires the creation of several new Py-T-Core rules. This section will describe the PositiveIndirectAssociation element and its associated Py-T-Core rules and the next will describe the NegativeIndirectAssociation elements and its associated Py-T-Core rules.

The PositiveIndirectAssociation represents a relationship between two classes where the target class has at least one relationship with itself. The rule will follow the associations and provide a match for each element along the route. To implement this functionality, the traversal along associations and subsequent matching is broken up into three rules that are used along with the actual rule specified in DSLTrans to correctly transform the model. The LHS and RHS of all the rules is sent to the new PositiveIndirectRule. The traversal itself uses pivots in the three rules to continue to process each branch along the indirect association. For clarity, an example of a PositiveIndirectRule with associated setup, initialize, and traverse rule is shown in Figure 9 using the GenealogyTree example introduced in the DSLTrans manual, along with a suggested change to the formalism to accommodate nested marriages. An example of the output of such a rule based on a given input model is shown in Figure 10. A ring corresponds to a marriage and a heart to a couple. A direct descendant relation is depicted by an solid arrow. A descendant relation of any depth is depicted by a smaller dashed arrow. A filled in stick figure corresponds to person in the genealogy tree meta-model. A stick figure outline corresponds to a person in the couple set meta-model. *Generic links* are used as explicit traceability links.

The first rule is the setup rule. The LHS of this rule takes everything from the DSLTrans ApplyModel except the part with the indirect association. If that can be found, a pivot labeled initial is set up on the target instance of the indirect association. In the Genealogy Tree to CoupleSet example, we only need to find instances of marriages from the input model as shown in the LHS

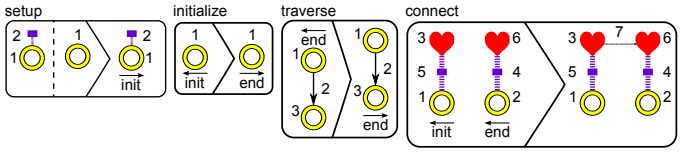



**Fig. 9** An example set of rules to check for a deep containment relationship to match every couple with its descendant couples.

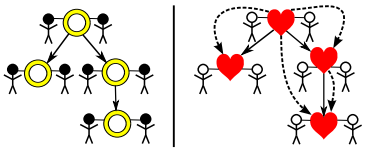


**Fig. 10** An example of an input model and the ouput descendant couples.

of the setup rule. The RHS of the rule will set up the pivot for each setup match found. It will also add a generic link from the marriage back to itself which the NAC checks to make sure the rule will not be applied again on the same element within the layer. This pivot is then used in the second rule, the initialize rule, to tie a pivot labeled `end` to the same instance as the initial pivot. In our example, the initialize rule matches the initial pivot marriage already created and adds the additional `end` pivot to the same marriage. The traverse rule will move along the associations to find further indirectly connected class instances by moving the end pivot. This is done using a depth-first search because the connections along associations could form a tree if the connection has cardinality of 0..*. The PositiveIndirectRule uses nested rule functionality to apply all four rules in the correct order. As we traverse, we need to apply the actual rule. In our example, we are connecting the couple associated with the initial marriage to every other couple associated with a marriage found during the traversal. Therefore, when we are finished, every couple should be connected to all of its descendant couples. The rule connect is therefore executed after every application of the traverse rule. The generic link between a marriage and a couple was added in a previous layer.

**Listing 6** A rule that tests for indirect containment before applying the actual rule in Py-T-Core.

```
class PositiveIndirectRule(Composer):
  def __init__(self, setupLHS, setupRHS, initialLHS, initialRHS,
        traverseLHS, traverseRHS, rule, max_iterations=INFINITY):
    super(PositiveIndirectRule, self).__init__()
    inner_traversal_rule = Sequence([rule,
      LSRule(traverseLHS, traverseRHS,rule, True, max_iterations)])
    outer_traversal_rule = Sequence([ARule(initialLHS, initialRHS),
                  IRule(traverseLHS, traverseRHS,
                        inner_traversal_rule, max_iterations)])
    self.M = Matcher(condition=setupLHS, max=max_iterations)
    self.I = Iterator(max_iterations=max_iterations)
    self.W = Rewriter(condition=setupRHS)
```

```
    self.inner_rule = outer_traversal_rule
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet) #Choose the 1st match
    if not self.I.is_success: return packet
    while True:
      packet = self.W.packet_in(packet)
      if not self.W.is_success: return packet
      packet = self.inner_rule.packet_in(packet) #Apply inner rule
      packet = self.I.next_in(packet)
      if not self.I.is_success: #No more iterations are left
        self.is_success = True #Output success packet
        return packet
```

As shown in the code of Listing 6, the `packet_in` method specifies the scheduling of these four rules. Note how the rule to be applied at every traversal step is given as a parameter (the connect rule in our case). This rule can be an arbitrary Py-T-Core Composer. But typically, an ARule is used if it is only using ExistsMatchClass elements in DSLTrans or an FRule if there is at least one AnyMatchClass element. The inner rule to handle traversal and rule application is a specific further nesting of rules. Since the outer setup rule is always applied first in an PositiveIndirectRule, the inner traversal and rule application will be correctly applied to every instance that could be matched as a valid item for setup.

The indirect rule's inner rule is the heart of the traversal process. Specifically, the inner rule is a pair of nested Sequence rules to handle a depth-first search. The outer traversal Sequence [the indirect rule's inner_rule] runs the initialize rule to prepare for traversal along the previously selected setup path. For each initialized path, it starts the traversal using an IRule. An IRule is a PositiveIndirectRule without a `rule` to be applied during the traversal. This traversal handles the breadth portion of the depth-first search. An example of the IRule can be seen in Algorithm 15. The initial run of the traverse rule will find the end pivot assigned in the initialize rule, and will move the end pivot to a connected indirect association. As mentioned, the Matcher has stored all of the other connected indirect associations from element with the previous end pivot on the packet, so when we eventually return to this point it will be able to choose the next match appropriately. Until then, the inner rule of the IRule starts. This inner rule is the second of the pair of nested Sequence rules mentioned earlier. Since we have just found a connected indirect association and labeled the appropriate class element with the end pivot, we can apply the `rule` to that endpoint. The first rule in the internal Sequence is the actual rule so it is applied now. The `rule` should consist of at least the model element labeled with the initial pivot. Once this first rewrite succeeds, we can proceed with the next set of traversals. This is handled by an LSRule which will apply an inner rule for each application of the outer rule as long as matches can be found. This is the same as the IRule except that it re-applies the outer rule on every iteration. This works in our favor because the traverse rule (the outer

rule for our usage of the LSRule) will continually move the end pivot down the indirect hierarchy in a depth-first manner as it is rematched. After each individual depth traversal, the inner rule will apply the actual rule on each new endpoint in turn so all matches will be found. The remaining matches will be found as we return to the outer IRule and follow other paths along the breadth of the containment tree. Note that if any of the actual traversals fail, the inner rules will never be applied so the actual rule will never occur when it does not meet the indirect containment relationship. Using the four discussed rules (setup, initialize, traverse, and rule) T-Core can handle any PositiveIndirectAssociation.

### 5.3 Negative Indirect Associations

While the PositiveIndirectAssociation specifies that an association must exist between elements, the NegativeIndirectAssociation specifies that the indirect association cannot exist between elements. The negative indirection may be dependent on an attribute on the NegativeMatchClass connected to the NegativeIndirectAssociation so we cannot tell if the association really failed until we traverse all the way down the hierarchy without finding a match for the the negative class with the specified attribute value. The negation aspect causes changes to the patterns introduced for the indirect association, but the basic structure is comparable. Listing 7 shows the negative indirect rule. It contains references to several negative versions of previously introduced rules. One major difference in this setup from the original one is the introduction of queries to test if the negative condition has been met. If it has, the NQuery reverses the value of is_success, so a failure to match looks like a success to the containing rule. The containing rules NIRule and NLSRule must also return success as true if their matcher or iterator fail, since if a traversal (the outer rule) failed, there is not even a direct link to the associated classes and the inner query will never be reached. The other change for the NIRule is to immediately return the packet if the inner rule fails. Since this is a negative test, if any inner rules fail there is no reason to continue matching since a counterexample has already been found. The only other change to NLSRule is to set the is_success state to false if an inner rule fails because even if the previous applications of the outer rules succeeded, the query failed so the entire rule should still fail.

The sequence here is similar to the positive case. The setup rule sets up the initial pivot, the initialize rule sets up the end pivot so traversal can begin, and the traverse rule actually moves along the containment associations. At each point, instead of applying a rule as in the positive version, the current model element associated with the end pivot is tested to see if it matches the undesired case. If it does, the query fails. Assuming the traversal for a given setup rule succeeds all the way down the containment hierarchy, the NegativeIndirectRule now applies the actual rule in the only major change from the `packet_in` of the PositiveIndirectRule. The five rules (four standard, one

query) handle any NegativeMatchAssociation assuming the actual rule once again uses the initial pivot and any needed connected components.

**Listing 7** A rule that tests for absense of indirect containment before applying the actual rule in Py-T-Core.

```
class NegativeIndirectRule(Composer):
  def __init__(self, setupLHS, setupRHS, initialLHS, initialRHS,
               traverseLHS, traverseRHS, negativeTestLHS,
               apply_to_all, max_iterations=INFINITY):
    super(NegativeIndirectRule, self).__init__()
    inner_traversal_rule = Sequence([
                        NQuery(negativeTestLHS),
                        NLSRule(traverseLHS, traverseRHS,
                            NQuery(negativeTestLHS),
                            True, max_iterations)])
    self.inner_rule = Sequence([ARule(initialLHS, initialRHS),
                  NIRule(traverseLHS, traverseRHS,
                        inner_traversal_rule, max_iterations)])
    self.M = Matcher(condition=setupLHS, max=max_iterations)
    self.I = Iterator(max_iterations=max_iterations)
    self.W = Rewriter(condition=setupRHS)
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet) #Choose the 1st match
    if not self.I.is_success: return packet
    while True:
      packet = self.W.packet_in(packet)
      if not self.W.is_success: return packet
      packet = self.inner_rule.packet_in(packet) #Apply inner rule
      if self.inner_rule.is_success:
        packet = self.actual_rule.packet_in(packet)
        if not self.actual_rule.is_success:
          self.is_success = False
          return packet
      packet = self.I.next_in(packet)
      if not self.I.is_success: #No more iterations are left
        self.is_success = True
        return packet
```

## 5.4 Validation

In this section we have shown how the full DSLTrans model transformation language is re-constructed in T-Core. Since no formal proof can endorse this claim, we have run a large amount of exhaustive tests to verify the equivalence between the two tools. For this purpose we have varied both the transformation domain meta-models (available with the DSLTrans documentation and implementation) and the size of the models. Furthermore, we considered a test as successful if it positively answered the following questions: Is the output model from Py-T-Core is syntactically the same as the the one output

from DSLTrans? Are the generated traceability links from DSLTrans a sub-set of the generic links produced by the Py-T-Core transformation, given specific input and output models? We currently do not have automatic support for answering these questions, therefore a manual inspection was required. In the future, we plan on using techniques from the model-based testing community, for example *mutation analysis* [40].

## 6 Related Work

There has been several proposals to leverage the complexity of using existing MTLs using a diversity of MTLs. For example Eclectic [12] consists of a family of MTLs each dedicated to a specific task. TransML [24] is stack of MTLs where lower level languages refine the upper ones from a development process point of view. EMFTVM [60] proposes a common virtual machine to implement ATL and other rewriting languages. In contrast, T-Core offers constructs at an intermediate level between the previous two. A specific combination of the operators, glued with an appropriate scheduling language leads to specific transformation languages.

In the context of global model management, the authors of [59] define a type system offering a set of primitives for model transformation. The advantage of our approach is that T-Core is described here as a module and is thus directly implementable. Also, the approach described in [59], does not deal with exceptions at all unlike T-Core. Nevertheless, their framework is able to achieve higher-order transformations (HOTs), *i.e.,* transformations that operate on model transformations. The implementation of T-Core is currently available in Python. Since this is an object-oriented language, the T-Core primitive operators are implemented as classes. Thus, at run-time, the operators are objects which can be directly manipulations and thus emulate HOTs. However, as mentioned in Section 4, T-Core can be combined with a modeling language. Thus, HOTs can be easily specified in such a completely modeled transformation language.

The GP graph transformation language [37] also offers transformation primitives. The authors 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 [57] and GReAT [1], have their own virtual machine used as an API. In our approach, since the primitive operations are modeled, they are completely compatible with other existing model transformation frameworks.

T-Core does cover a significant amount of variation in pattern-based model transformation. For example, we showed how to solve the amalgamated rule

problem where pattern elements are combined with universal and existing quantifiers. This was done by wisely "nesting" pre-condition patterns with the use of pivots. Other pattern compositions include disjunctive constructs such as in [4]. That is, a LHS pattern can consist of sub-patterns that can be conjuncted and disjuncted. This can be accomplished with T-Core primitives as illustrated in chapter 10 of [49]. When the LHS consists of two disjuncted patterns, we first split each disjunctive case in separate pre-condition patterns. Then, the `packetIn` method of the Matcher of each pattern is called. Each resulting packet is output to a Selector which finally selects one of the packets.

The detection of conflicts in the Resolver and the Synchronizer is currently conservative. However the user can override the detection with transformation- or model-specific resolution and merging algorithms. The exception handling mechanism in place can also be used for optimistic resolution [51]. An alternative is to incorporate more advanced detection mechanisms, such as through critical pair analysis [33]. However, this technique assumes that the transformation units are traditional graph transformation rules with a single *match-rewrite* combination, which is not always the case in T-Core.

## 7 Conclusion

This article motivated the need for providing MTL primitives. T-Core was defined by precisely describing each of these primitive constructs. The deconstruction process of MTLs enabled us to re-construct existing simple model transformation features as well as more complex ones by combining T-Core with, for example, an SBL language. This allowed us to compare different MTLs using a common basis. Furthermore, T-Core is combined with a programming language which allows non-MDE users to integrate with MDE solutions. This integration is transparent for programmers since Py-T-Core and T-Core offer a complete API.

T-Core was presented as a minimal collection of model transformation primitives, defined at the optimal level of granularity. It is not restricted to any form of specification of transformation units, be it rule-based, constraint-based, or function-based. It can also represent bidirectional and functional transformations as well as queries. T-Core modularly encapsulates the combination of these primitives through composition, re-use, and a common interface. It is an executable module that is easily integrable with a programming or modeling language.

It is impossible to prove that T-Core is a collection of the most primitive transformation operators, because of the complexity and diversity of the expressiveness of most MTLs. Nevertheless we have illustrated throughout this article how complex transformation units particular to different existing transformation language can be defined in T-Core. However declarative transformations defined as relations, such as in QVT-Relation or Triple Graph Grammars, cannot be directly expressed using T-Core primitives. That is because their transformation units specify relations between the involved meta-models

as opposed to the operational nature of transformation rules. However, if these relations can be compiled into operational rules such as in [46], then T-Core primitives can be used to mimic the corresponding behavior of the relations.

T-Core can serve as a basis for inter-operating model transformations expressed in different formalisms. That is, by mapping each and every construct of the languages to an appropriate combination of T-Core operators. In [25], the authors define a language for composing heterogeneous transformations defined in different formalisms (*e.g.,* ATL and QVT Operational Mappings). Their approach is to wrap each transformation model in *components* and communicate between each other via in/out-port connections, treating the transformation models as black-boxes. This is the opposite of opening the languages and mapping them to a common denominator: T-Core. The disadvantage of their approach is that port connection consistency is validated through simple type checking. Also, their current implementation is restricted to models only represented in Ecore.

We are currently working on techniques to analyze transformation languages built with T-Core as initiated in [50]. Also, the process of mapping different transformation has lead to the discovery of re-usable complex transformation units. We plan to generalize them and build a catalog of such "transformation language design patterns".

## References

1. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal on Software and Systems Modeling **5**(3), 261–288 (2006)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: D. Petriu, N. Rouquette, Ø. Haugen (eds.) Model Driven Engineering Languages and Systems, *LNCS*, vol. 6394, pp. 121–135. Springer (2010)
3. ATLAS group, LINA, INRIA Nantes: Specification of the ATL Virtual Machine. http://www.eclipse.org/m2m/atl/doc/ (2005)
4. Balogh, A., Varró, D.: Pattern Composition in Graph Transformation Rules. In: European Workshop on Composition of Model Transformations. Bilbao (2006)
5. Barroca, B., Lúcio, L., Amaral, V., Felix, R., Sousa, V.: DSLTrans: A Turing Incomplete Transformation Language. In: International Conference on Software Language Engineering, LNCS. Springer, Eindhoven (2010)
6. Batz, G.V., Kroll, M., Geiß, R.: A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In: A. Schürr, M. Nagl, A. Zündorf (eds.) International Symposium on Applications of Graph Transformations with Industrial Relevance, *LNCS*, vol. 5088, pp. 471–486. Springer (2008)
7. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental Pattern Matching in the VIATRA Model Transformation System. In: International Workshop on Graph and Model Transformation (2008)
8. Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM **9**(5), 366–371 (1966)
9. Jeroen van den Bos, Tijs van der Storm: Domain-Specific Optimization in Digital Forensics. In: J. de Lara, Z. Hu (eds.) Theory and Practice of Model Transformation, *LNCS*, vol. 7307, pp. 121–136. Springer, Prague (2012)
10. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions On Pattern Analysis and Machine Intelligence **26**(10), 1367–1372 (2004)

11. Csárdi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal Complex Systems **1695** (2006). URL igraph.sourceforge.net
12. Cuadrado, J.S.: Towards a Family of Model Transformation Languages. In: J. de Lara, Z. Hu (eds.) Theory and Practice of Model Transformation, *LNCS*, vol. 7307, pp. 176–191. Springer, Prague (2012)
13. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, special issue on Model-Driven Software Development **45**(3), 621–645 (2006)
14. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of graph grammars and computing by graph transformation, Volume 1: Foundations. World Scientific Publishing Co., Inc. (1997)
15. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (eds.) International Conference on Graph Transformation, *LNCS*, vol. 3256, pp. 161–177. Springer-Verlag, Rome (2004)
16. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. In: A. Zündorf, D. Varró (eds.) International Workshop on Graph Based Tools, *ECEASST*, vol. 1, pp. 1–13. Natal (2006)
17. Fischer, T., Niere, J., Turunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In: H. Ehrig, G. Engels, H.J. Kreowski, G. Rozenberg (eds.) Theory and Application of Graph Transformations, *LNCS*, vol. 1764, pp. 296–309. Springer-Verlag, Paderborn (2000)
18. Fulgham, B.: Computer Language Benchmarks Game (2010)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional (1994)
20. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.) International Conference on Graph Transformation, *LNCS*, vol. 4178, pp. 383–397. Springer-Verlag, Heidelberg (2006)
21. Gomes, C., Barroca, B.: DSLTrans Manual (2011)
22. Object Management Group: Unified Modeling Language Superstructure, 2.2 edn. (2009). URL http://www.omg.org/spec/UML/2.2/Superstructure/PDF/
23. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation Specification (2011)
24. Guerra, E., de Lara, J., Kolovos, D., Paige, R., dos Santos, O.: Engineering model transformations with transML. Software and Systems Modeling **in press**, 1–23 (2011)
25. Heidenreich, F., Kopcsek, J., Assmann, U.: Safe Composition of Transformation. In: L. Tratt, M. Gogolla (eds.) Theory and Practice of Model Transformation, *LNCS*, vol. 6142, pp. 108–122. Springer-Verlag, Màlaga (2010)
26. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop, *LNCS*, vol. 3844, pp. 128–138. Springer-Verlag (2006)
27. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons (2008)
28. Knirsch, P., Kuske, S.: Distributed Graph Transformation Units. In: A. Corradini, H. Ehrig, H.J. Kreowski, G. Rozenberg (eds.) International Conference on Graph Transformation, *LNCS*, vol. 2505, pp. 207–222. Springer, Barcelona (2002)
29. Königs, A., Schürr, A.: MDI: A Rule-based Multi-document and Tool Integration Approach. Journal on Software and Systems Modeling **5**(20), 349–368 (2006)
30. Krissinel, E.B., Henrick, K.: Common subgraph isomorphism detection by backtracking search. Software - Practice & Experience **34**(6), 591–607 (2004)
31. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Systematic Transformation Development. Electronic Communications of the European Association of Software Science and Technology **21** (2009)
32. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit Transformation Modeling. In: S. Ghosh (ed.) MODELS 2009 Workshops, *LNCS*, vol. 6002, pp. 240–255. Springer, Denver (2010)
33. Lambers, L., Ehrig, H., Orejas, F.: Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs. In: R. Bruni, D. Varró (eds.) International Workshop on Graph Transformation and Visual Modeling Techniques, *ENTCS*, vol. 211, pp. 17–26. Vienna (2008)

34. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. Journal of Visual Languages and Computing **15**(3–4), 309–330 (2004)
35. LaShomb, B., Syriani, E.: Re-engineering DSLTrans with T-Core. Tech. Rep. SERG-2012-04, University of Alabama, Department of Computer Science (2012)
36. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model Transformation with a Visual Control Flow Language. International Journal of Computer Science **1**(1), 45–53 (2006)
37. Manning, G., Plump, D.: The GP Programming System. In: C. Ermel, R. Heckel, J. de Lara (eds.) International Workshop on Graph Transformation and Visual Modeling Techniques, *ECEASST*, vol. 10, pp. 235–247. Budapest (2008)
38. Mehlhorn, K.: Graph Algorithms and NP-Completeness, *Monographs in Theoretical Computer Science. An EATCS Series*, vol. 2. Springer (1984)
39. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. In: International Workshop on Graph and Model Transformation, *ENTCS*, vol. 152, pp. 125–142. Tallinn (2006)
40. Mottu, J.M., Baudry, B., Le Traon, Y.: Mutation Analysis Testing for Model Transformations. In: European Conference on Model Driven Architecture: Foundations and Applications, pp. 376–390 (2006)
41. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: International Conference on Software Engineering, pp. 742–745. ACM Press, Limerick (2000)
42. Eclipse Modeling Project: ATL Transformations. http://www.eclipse.org/m2m/atl/atlTransformations (2010)
43. Provost, M.: Himesis: A Hierarchical Subgraph Matching Kernel for Model Driven Development. Master's thesis, McGill University, Montréal (2005)
44. Rensink, A., Kuperus, J.H.: Repotting the Geraniums: On Nested Graph Transformation Rules. In: T. Margaria, J. Padberg, G. Taentzer (eds.) International Workshop on Graph Transformation and Visual Modeling Techniques, *ECEASST*, vol. 18. York (2009)
45. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: H. Ehrig, G. Engels, H.J. Kreowski, G. Rozenberg (eds.) TAGT'98, Selected Papers, *LNCS*, vol. 1764, pp. 381–394. Springer, Paderborn (1998)
46. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars. In: H. Ehrig, R. Heckel, G. Rozenberg, G. Taentzer (eds.) International Conference on Graph Transformation, *LNCS*, vol. 5214, pp. 411–425. Leicester (2008)
47. Schürr, A., Winter, A.J., Zündorf, A.: Graph Grammar Engineering with PROGRES. In: W. Schäfer, P. Botella (eds.) European Software Engineering Conference, *LNCS*, vol. 989, pp. 219–234. Springer-Verlag, Sitges,Spain (1995)
48. Syriani, E.: T-Core. http://msdl.cs.mcgill.ca/people/eugene/motif/tcore.zip (2010)
49. Syriani, E.: A Multi-Paradigm Foundation for Model Transformation Language Engineering. Ph.d. thesis, McGill University (2011)
50. Syriani, E., Gray, J.: Challenges for Addressing Quality Factors in Model Transformation. In: First International Workshop on Verification of Model Transformation. Montreal (2012)
51. Syriani, E., Kienzle, J., Vangheluwe, H.: Exceptional Transformations. In: L. Tratt, M. Gogolla (eds.) Theory and Practice of Model Transformation, *LNCS*, vol. 6142, pp. 199–214. Springer-Verlag, Màlaga (2010)
52. Syriani, E., Vangheluwe, H.: Matters of model transformation. Tech. Rep. SOCS-TR-2009.2, McGill University, School of Computer Science (2009)
53. Syriani, E., Vangheluwe, H.: DEVS as a Semantic Domain for Programmed Graph Transformation, chap. 1, pp. 3–28. CRC Press, Boca Raton (2010)
54. Syriani, E., Vangheluwe, H.: Performance Analysis of Himesis. Tech. Rep. SOCS-TR-2010.8, McGill University, School of Computer Science (2010)
55. Syriani, E., Vangheluwe, H.: A Modular Timed Model Transformation Language. Journal on Software and Systems Modeling **11**, 1–28 (2011)
56. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. Journal of the ACM **23**(1), 31–42 (1976)
57. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming **68**(3), 214–234 (2007)

58. Varró, G., Varró, D., Friedl, K.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In: G. Karsai, G. Taentzer (eds.) International Workshop on Graph and Model Transformation, *ENTCS*, vol. 152, pp. 191–205. Elsevier, Tallinn (2005)
59. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in Model Management. In: R. Paige (ed.) Theory and Practice of Model Transformation, *LNCS*, vol. 5563, pp. 197–212. Springer-Verlag, Zürich (2009)
60. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a General Composition Semantics for Rule-Based Model Transformation. In: J. Whittle, T. Clark, T. Kühne (eds.) Model Driven Engineering Languages and Systems, *LNCS*, vol. 6981, pp. 623–637. Springer, Wellington (2011)
61. Wimmer, M., Kusel, A., Schönböck, J., Reiter, T., Retschitzegger, W., Schwinger, W.: Lets's Play the Token Game – Model Transformations Powered By Transformation Nets. In: Workshop on Petri Nets and Software Engineering, pp. 35–50. Université Paris 13, Paris (2009)
62. Zündorf, A.: Graph Pattern Matching in PROGRES. In: H. Ehrig, G. Engels, G. Rozenberg (eds.) Graph Grammars and Their Application to Computer Science, *LNCS*, vol. 1073, pp. 454–468. Springer-Verlag, Williamsburg (1996)