Explicit Transformation Modeling

Thomas Kühne[1], Gergely Mezei[2], Eugene Syriani[3], Hans Vangheluwe[3,4], and Manuel Wimmer[5]

[1] Victoria University of Wellington
tk@ecs.victoria.ac.nz
[2] Budapest University of Technology and Economics
gmezei@aut.bme.hu
[3] McGill University
{esyria,hv}@cs.mcgill.ca
[4] University of Antwerp
hans.vangheluwe@ua.ac.be
[5] Vienna University of Technology
wimmer@big.tuwien.ac.at

Abstract. Despite the pivotal significance of transformations for model-driven approaches, there have not been any attempts to explicitly model transformation languages yet. This paper presents a novel approach for the specification of transformations by modeling model transformation languages as domain-specific languages. For each pair of domain, the metamodel of the rules are (quasi-)automatically generated to create a language tailored to the transformation. Moreover, this method is very efficient when the transformation domains are the transformation rules themselves, which facilitates the design of higher-order transformations.

1 Introduction

Model-driven approaches are gaining popularity both in the form of being based on standard modeling languages, such as the UML, as well as domain-specific modeling languages. In both instances, the aim is to increase developer productivity, in the case of the former by raising the level of abstraction at which systems can be specified and in the case of the latter by lowering the impedance mismatch between a modeling language and its application domain [1].

There are still many open problems with respect to the economic development of domain-specific modeling languages, but their definition is well understood. This shifts the focus on transformations which have a number of applications among which are: (1) establishing transformation chains from high-level to low-level specifications, (2) providing semantics for a source language by mapping it to a target language, and (3) creating a consistent mapping between two or more models. A number of transformation paradigms exists, e.g., template-based, rule-based, triple graph grammars, with or without explicit control flow [2]. They are supported by various implementations such as ATL [3], AToM^3 [4], GReAT [5], Moflon [6], QVT [7], VMTS [8]. They provide tremendous value for developers, but in each implementation the transformation paradigm is hard-coded to be

S. Ghosh (Ed.): MODELS 2009 Workshops, LNCS 6002, pp. 240–255, 2010.

used as is. The implementations do not provide a way to interrogate or modify transformation definitions as first-class transformation models [9]. This is surprising as there are a number of benefits to be gained when treating transformations as first-class citizens [10,11] which are explicitly modeled and amenable to introspection and modification.

In the following, we first introduce our running example which we use as the basis of our subsequent discussions. In Sect. 3, we investigate the automated construction of customized pattern specification languages, using the components relaxation, augmentation, and modification, exploring and discussing alternative solutions. This provides a systematic *procedure* for explicitly modeling transformation languages. As a consequence, this enables to "cleanly" design higher-order transformations. In Sect. 4, we present two higher-order transformations – for automatically enhancing transformations with correspondence links and adding source-level animation for simulations respectively – which exhibit two different forms of separation of transformation concerns. Before we conclude, we compare our transformation definitions with those required in an ATL context and discuss further related work in Sect. 5.

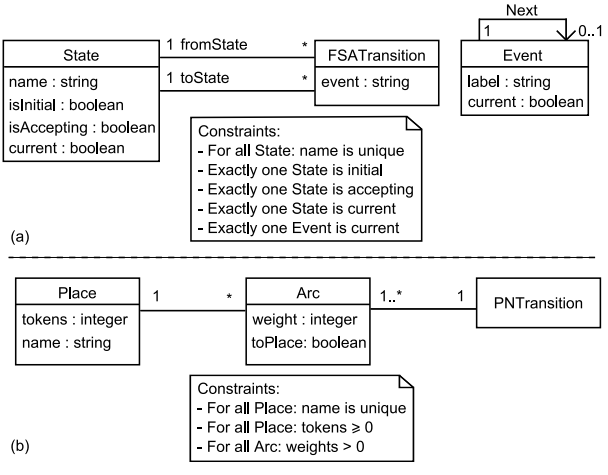

Fig. 1. (a) FSA & (b) Petri Net Metamodels

2 Running Example

The example that we will use in the remainder of the paper to illustrate our arguments is a typical case of a domain-specific language being assigned a semantics by translating it into a target formalism with known semantics. In order to define the semantics of statecharts and/or perform reachability analyses on them, one can translate them to Petri nets [4]. Another reason for considering this particular translation is that one can use Petri nets as a common semantic

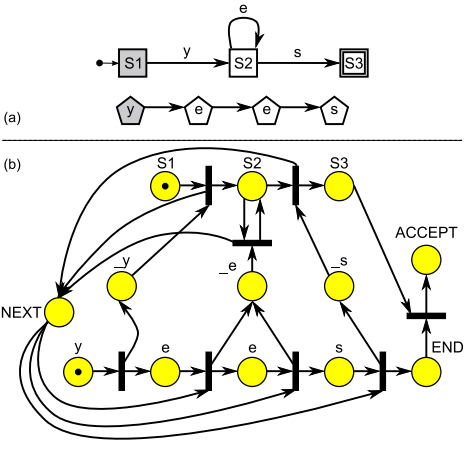

Fig. 2. (a) FSA to (a) Petri Net Mapping

domain for statecharts, sequence diagrams, and activity diagrams. For the purposes of this paper, however, we restrict ourselves to translating finite state automata, rather than statecharts, into Petri nets. The resulting transformation definitions of this translation are much simpler but still rich enough to illustrate our arguments. Figure 1 shows both metamodels.

2.1 Finite State Automata as Language Recognizers

More specifically, we interpret our state automata to be language recognizers, i.e., they either accept input sequences as belonging to respective regular languages or not. The top part of Fig. 2 shows a sample input sequence ("yees") and a finite state automaton accepting the language $y(e)^*s$. In our example, we want to simulate the execution of the finite state automaton in the context of receiving the events from the input sequence in order to ascertain whether the input sequence is a sentence of the language. To this end, we translate such scenarios into corresponding Petri nets (see bottom part of Fig. 2).

2.2 Translating Finite State Automata to Petri Nets

Figure 3 shows an excerpt of the transformation rules that are required to translate a finite state automaton plus an input sequence into a Petri net that can be used to simulate the automaton execution. In particular, Figure 3 shows a subset of the rules that translate finite automaton states into Petri net places. The rest of the rules dealing with the translation are similar to those shown and are not of further significance for the purposes of this paper. Note that we are using AToM^3 [4] and MoTif [12] and thus use concrete syntax for describing single push-out graph transformation rules, employing numerical labels to indicate identity of elements.

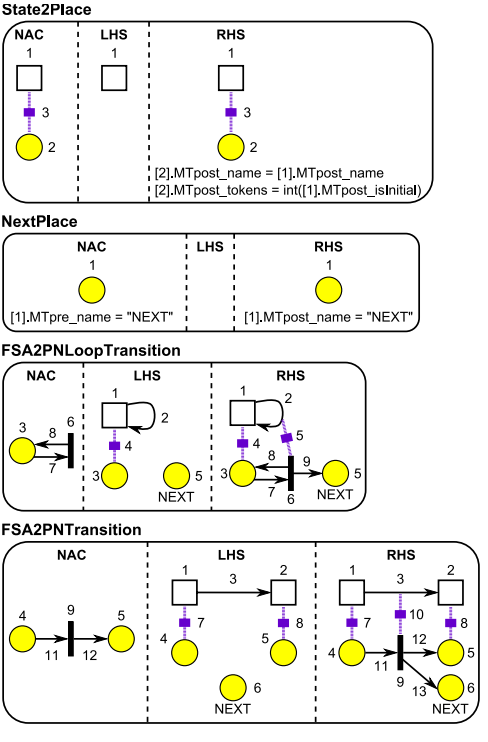

Fig. 3. Translating States to Places

We are well aware of the tension between the "must transition" and "may fire" semantics of finite state automata and Petri nets, respectively. In timed Petri nets, this difference may lead to a situation where a finite state automaton does not change states anymore even though it *should*, just because the Petri net used for simulating it does not fire transitions anymore, even though it *could*. However, the

place/transition nets we assume do not create this mismatch and a simulator for them will fire enabled transitions.

2.3 Petri Net Semantics

We simulate Petri net execution by using a small set of transformation rules, from Petri nets to Petri nets, which realize an operational semantics of Petri nets (see Fig. 4). We are able to express the operational semantics in just four simple rules because we use MoTif (Modular Timed graph transformation language) control structures, which are based on discrete event-based control structures [12].

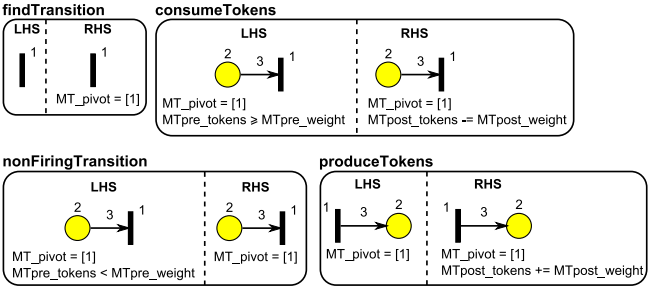

Fig. 4. Operational Semantics for Petri Nets

The respective control structure is shown in Fig. 5. The control structure shown in Fig. 5 makes it particularly easy to find an enabled Petri net transition, i.e., one which can fire. Such a transition needs sufficiently many tokens at *each* of its incoming transitions. One naive solution for finding enabled transitions is to just specify all possible patterns to be found as subgraph isomorphisms. Alternatively, this can be solved provided that the pattern specification language uses intentional specifications to allow referring to subgraphs of arbitrary size. However, the most elegant solution is to iterate through all transitions until one has been found that does *not* satisfy the pattern of a *non*-firing transition. The backward channel in Fig. 5 from the *success* port (depicted by a check mark) of NonFiringTransition to the *next* port (depicted by two filled triangles) of FindTransition ensures the iteration through all transitions, skipping those which cannot fire. Hence the forward channel from the *fail* port (depicted by a cross) of NonFiringTransition to the *graph* port (depicted by a triangle) of the FireTransition block.

The latter's content are the two rightmost rules of Fig. 4. They are applied to all matching patterns within the subgraph that has been passed to FireTransition. Circles around output ports indicate that a pivot model (matched model element) is passed to the next port (i.e., *next* and *fail* ports). Therefore, FireTransition will fire exactly the transition which has been first found by FindTransition and subsequently has not been rejected by NonFiringTransition.

In Sect. 4, we will use a higher-order transformation to extend the control structure shown in Fig. 5 to include an animation component. First, however, we will describe and discuss the explicit modeling of transformation definitions as an enabler of customized transformation development environments and higher-order transformations.

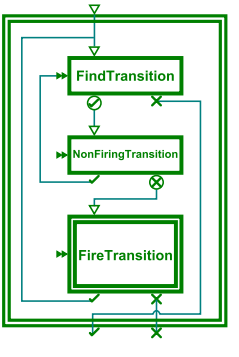

Fig. 5. Semantics Control Structure

3 Explicit Transformation Modeling

Metamodeling[1], i.e., the explicit specification of a language's well-formedness constraints, has become popular because of a number of associated advantages: (1) the specification is not hidden in the code of a tool, making it easier to understand and correct, (2) the specification can be altered by users of the tool instead of requiring a new tool release, and (3) one can reason about the specification and the models it describes. The same advantages apply if metamodeling is not only applied to modeling language definitions, but also to transformation definitions. While there is a considerable initial investment to be made in explicitly modeling a transformation language including its semantics, the prospect to more easily experiment with language features, customize them for certain purposes, and allow transformations to be reasoned about and/or modified makes that investment worthwhile.

Clearly, in order to enable the last aspect mentioned above, the transformation language's mapping approach, e.g., rule-based graph transformation, needs to be explicitly modeled. Section 4 elaborates on this and additionally motivates why a transformation language's control part should also be explicitly modeled.

Unlike the mapping and control aspects of a transformation language, its pattern specification sublanguage depends on other languages. The input and output languages of a transformation determine which pattern specifications for left-hand side (LHS) and right-hand side (RHS) can be considered well-formed. The underlying assumption here is that the pattern specification language should not be generic to fit all possible input and output languages, but specifically tailored to the input and output languages involved.

3.1 Generic versus Customized Pattern Specification Languages

The most economic approach to providing a pattern specification language is to offer a generic one. Most tools do not use concrete syntax for specifying transformation patterns and thus are able to use the same generic (often UML object-diagram-inspired) pattern specification syntax for all possible input/output languages. They often also have an underlying generic (often MOF-like) representation format which can be used to represent elements from any input/output language.

There are good reasons, however, to consider using a pattern specification language which is customized to the input/output languages involved:

- One may use pattern specification visualizations which are adapted to the languages involved. Even if no concrete syntax is used, one may still want to customize the syntax, e.g., to adequately visualize connector elements.
- A customized syntax allows excluding patterns from being specified that do not have a chance of matching subgraphs in the host graphs. For instance, in the context of Petri nets, a pattern consisting of an arc linking two places will never be matched on any valid Petri net instance (i.e., conforming to the meta-model in Figure 1).

[1] Linguistic metamodeling [1], to be precise.

A generic pattern specification language will allow any pattern to be expressed whether or not it will be able to match subgraphs from the input language(s) or generate subgraphs conforming to the metamodel(s) of the output language(s). Just as a plain domain-specific modeling tool has advantages for its users, guiding them to produce meaningful models, a customized transformation pattern specification tool also aids in avoiding meaningless pattern specifications.

Whether this customization is achieved by changing the representation format for each generated transformation definition environment or by just exchanging a language definition against which generic pattern specifications are checked is immaterial to the user, but a tool builder decision. In the following, we assume that, in one way or another, pattern specifications can be checked for conformance to a pattern specification language definition. As a result, a method needs to be identified that enables these conformance checks in an economic manner, while offering the transformation language user maximum benefits.

3.2 Metamodels versus Conformance Checks

Unfortunately, providing a customized pattern specification language is not as easy as simply reusing the corresponding input/output metamodels. First, demanding a full adherence of pattern specifications to original language definitions is not practical. If all minimal multiplicity requirements of language definitions were enforced, one could not specify useful patterns such as findTransition of Fig. 4, which refer to model fragments, ignoring minimal multiplicity requirements. Second, one may want to provide several levels of rigor with respect to checking the well-formedness of pattern specifications. While the transformation designer edits a pattern specification, one most certainly does not want to enforce all well-formedness constraints. It also should be possible to save ill-formed sketches to be worked on later. This does not mean, however, that the complete absence of all potential well-formedness checks is always the best choice in such cases. Table 1 lists potentially useful levels of conformance checking rigorousness. There are two ways to enable the use of such levels of conformance:

1. either one creates modified language definitions and performs a normal conformance check against them, or
2. one uses original language definitions, but accordingly modified conformance checks.

Table 1. Levels of Conformance

Level of rigor	**Description**
Free form	no constraints at all
Valid elements	elements are typed by the metamodel
Valid multiplicities	(relaxed) multiplicity constraints are enforced
Valid constraints	(a subset of) metamodel constraints are enforced

The second option has a number of advantages:

- one can simply use the original language definitions; there is no need to create multiple variants of them.
- switching between conformance levels does not require the switch of a metamodel; the latter is quite feasible though with an appropriate architecture.
- the alternative (1. above) cannot use a standard conformance check anyhow (see Sect. 3.3 and Sect. 5).

However, there are also a number of disadvantages:

- some generic way to extend languages defined by metamodels is required; pattern specification languages require additional features beyond the original input/output languages (see Sect. 3.3). Customized metamodels can easily incorporate these.
- custom conformance checks are harder to reason about than custom metamodels; in the absence of a fully modeled action language, conformance checks will be implemented in some programming language making it harder to see and analyze what relation they actually implement.
- conformance checks are harder to customize by users; transformation designers can be expected to alter the transformations that yield tailored metamodels but may not be able to re-program conformance checks.
- swapping conformance checks means that the transformation development will remain the same; swapping metamodels opens up the possibility to use them for the automated generation of dedicated development environments with differing sets of control elements.

Finally, there is another motivation for supporting more than one mode of well-formedness checking which can only be enabled by using multiple metamodel versions: Typically, transformation definitions comprise layers of rules in the sense that one will expect all rules from one layer to have matched, and then match no more, before the next layer of rules will be used. This layering often exists independently of whether or not it is dealt with explicitly. In particular with in-place transformations, the input and output languages change from layer to layer. The first layer's input language is the source language while its output, the input to the next layer, will typically contain generic links which are not part of the source language (see Sect. 3.3). The last layer's output language is the target language, whereas all preceding layers will produce either augmented versions of it or mixtures between the source and target languages. The availability of a series of adapted metamodels may aid the transformation developer to understand what the layers involved are and assign rules to them accordingly.

We have not yet pursued the idea of using a series of transformation layer interface language definitions and it would be challenging to automate the generation of these intermediate language definitions. Luckily, however, automating the creation of customized pattern specification languages from original input/output language definitions can be automated very well.

3.3 Semi-automated Metamodeling

The previous section motivated the use of variants of original metamodels for defining the well-formedness of pattern specifications. In this section, we discuss how one can create such variants systematically and thus automate the process.

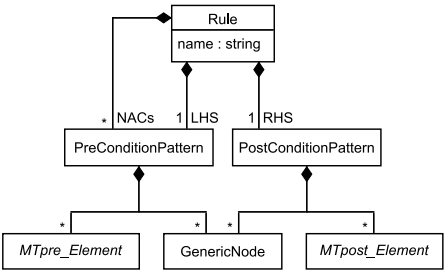

Fig. 6. Rule Metamodel

Figure 6 depicts how rules refer to precondition and postcondition patterns and the pattern element they contain. When adapting transformation languages to specific input and output languages, one needs to tailor these precondition and postcondition patterns so that they are fit to be used for the respective input and output languages. We obtain the required tailored pattern specification metamodels by starting with the original language metamodels and then subjecting them to a number of changes. The required metamodel metamorphosis has three distinct components: relaxation, augmentation, and modification. Figure 7 shows an excerpt of the result of applying these steps to the finite state machine metamodel of Fig. 1.

Relaxation

Original language definitions cannot be used as is for defining the well-formedness of pattern specifications. First, often transformation designers aim to match for any one-of-many element types, e.g. one-of-many "connection" kinds. Such generalizations are typically present in original language definitions but as abstract concepts which cannot be instantiated. One relaxation step therefore is to turn such abstract concepts into concrete ones.

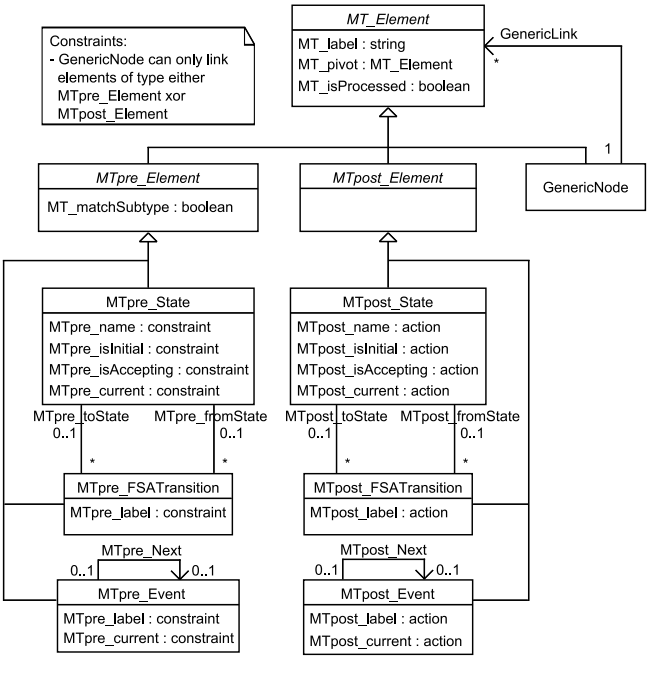

Fig. 7. Generated Pattern Specification Metamodel

Second, as mentioned before, enforcing minimal multiplicity constraints would be completely impractical. A further relaxation step is, therefore, to reduce all minimal multiplicities to zero (see Fig. 7 for the relaxation of State multiplicities and Sect. 5 for a more elaborate discussion).

Third, only a subset of explicitly formulated original constraints (e.g., using OCL) can be active for the purpose of checking pattern specification well-formedness. All constraints concerned with ensuring completeness of models are potentially unsuitable for the inherent fragment-like nature of specification patterns. The relaxation process could automatically filter out constraints with the help of a corresponding naming scheme for constraints or manually provided augmentations, but we currently believe any further automation will be difficult to achieve. This is why we refer to the metamodel generation as *semi*-automated.

A potential further relaxation is to raise all maximum multiplicities to "unbounded" in order to allow intermediate results that can be helpful to drive the transformation process, despite the fact that they would be ill-formed as end results. However, we argue that purposefully violating well-formedness requirements in this way amounts to "hacking" and should be avoided. We recommend using so-called generic links for these purposes instead.

Augmentation
To be fit as pattern specification metamodels, input/output metamodels also need to be augmented with features required for transformation purposes.

In Fig. 7, all types are made descendants of *MT_Element* so that they inherit features that all elements that may appear in a pattern specification must have, e.g., a way to label them for identity matching. The generated metamodels also feature additional generic nodes and links which are often necessary to drive the transformation (e.g., see the generic connectors between states and places in Fig. 3). Elements which are used in negative application conditions (NACs) or the LHS of rules (subtypes of *MTpre_element*) also need a flag feature that tells the pattern matcher whether to look for exact types or allow subtype matching as well. The remaining differences between the original and generated metamodel elements are all modifications of existing features.

Modification
The modifications that need to be applied to original metamodel elements depend on whether we want to obtain precondition (i.e., NAC and LHS) or postcondition (i.e., RHS) pattern specifications.

For precondition pattern specifications we need to replace the respective types of attributes to the type "constraint". This allows the transformation designer to specify constraints for element features, such as MTpre_name="NEXT" (see Fig. 3). For postcondition pattern specification we need to allow actions rather than constraints, so that the transformation designer can set values of attributes, among other potential actions. In rule NextPlace of Fig. 3, the "=" in the RHS part of the rule is an assignment action rather than an equality check. Note that the same naming and modification scheme is applied to classes, associations, and role names.

Finally, we sometimes need to modify the concrete syntax of language elements whose size or natural layout is not conducive for specifying patterns. Also, elements which are normally not rendered at all, such as instances of formerly abstract classes or association ends, need to be assigned some concrete syntax so that they may be referred to in a visual manner.

We have implemented a prototype of this procedure. A new metamodel is created as partly shown in Figure 7. In the relaxation step, we did not consider the (OCL) constraints of the respective metamodels yet and they thus have been maintained. In the augmentation step, the first two levels of the inheritance hierarchy of Figure 7 correspond to concepts from the meta-metamodel of ATOM^3/MOTIF. Finally in the modification step, our prototype did not take into account issues related to layout in the concrete syntax of the pattern elements.

Summarizing, this section has discussed various alternatives for enabling transformation designers to make use of customized pattern specification languages and environments. We proposed the semi-automated generation of customized metamodels based on the components of relaxation, augmentation, and modification.

Figure 8 depicts how a transformation from model $M1$ to model $M2$ is defined with this approach. Following the Finite State Automata to Petri Nets example, we call T_{FSA-PN} the (transformation) model mapping M_{FSA} to M_{PN}, respectively the models depicted by Fig. 2 (a) and (b). The two models conform to their metamodels MM_{FSA} and MM_{PN}, respectively. Applying the technique described in this section, *domain-specific pattern languages* are generated from these metamodels, namely PL_{FSA} and PL_{PN} respectively. The metamodels of the patterns (specific to this transformation) combined with the metamodel of the transformation control logic language, in our example MOTIF, form the *transformation language* TL_{FSA-PN}. The transformation T_{FSA-PN} is thus a model conforming to its metamodel TL_{FSA-PN}. One of the big advantages of such explicit transformation language modeling is the possibility to easily define higher-order transformations.

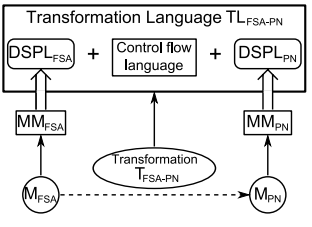

Fig. 8. Schema of domain-specific transformation languages

4 Higher-Order Transformations

There is ample motivation for transforming transformations by means of higher-order transformations. Promising application areas include:

Language evolution: Whenever the definition of a language evolves, any associated transformations have to be adapted. This adaptation process may sometimes be semi-automated by using a higher-order transformation generated from the modifications made to the language.

Transformation optimization: Optimizing transformations could modify transformations so that they and/or their results are more efficient.

Transformation definition: In some cases, one can obtain a translational semantics from an operational semantics by use of a higher-order transformation, so that one has ease of definition plus advanced analysis opportunities [13]. One may also define the meaning of transformations using high-level constructs by mapping them onto standard transformations [14].

Separation of transformation concerns: Instead of putting all functionality into one transformation, one can split concerns over many transformations and integrate them by sequentially adding them with higher-order transformations to a base transformation. Often one may use multi-stage transformations for the same effect, but sometimes this is not a viable option (see Sect. 4.2). Separating transformation concerns from each other does not only reduce the complexity of an otherwise monolithic transformation but also opens up the opportunity to reuse (higher-order) transformations.

4.1 Source-Level Animation

In conjunction, the "Finite State Automaton to Petri Net" and operational Petri net semantics transformations presented in Sect. 2 simulate a language recognizer but without visualizing the execution at the source level of state automata. In order to add source-level animation, we need to add update rules (see Fig. 9) to the operational Petri net semantics. AToM^3 then automatically takes care of updating the respective concrete syntax.

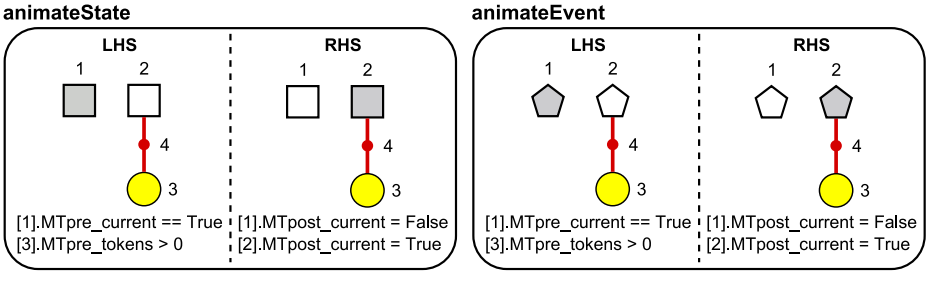

Fig. 9. Animation Rules

Having explicitly modeled all artifacts, we are now in a position to define a higher-order transformation which automatically adds the update rules to the operational semantics transformation. Block "1" at the bottom of the LHS pattern in Fig. 10 is a parameter to the higher-order transformation AddAnimation and contains all the update rules of Fig. 9. We perform the parameter passing by using the block as a pivot model. The RHS of higher-order transformation AddAnimation simply links the update block into the main loop of the original semantics control structure (see Fig. 5), making sure this happens only once at the top level (hence the NAC).

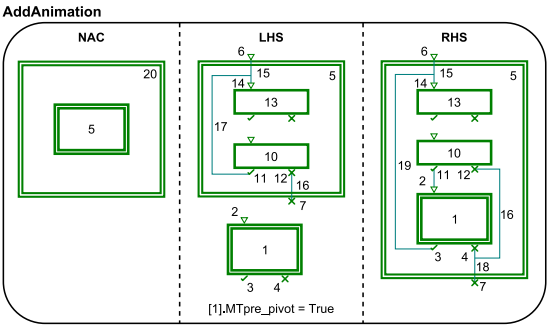

Fig. 10. H-O Transformation: Animation

Figure 11 shows the result of applying higher-order transformation AddAnimation to the original control flow shown in Fig. 5. Note that being able to modify an explicit representation of a transformation's control structure allowed us to design AddAnimation in a way that makes it reusable. As long as one designs other operational semantics definitions with a similar overall main loop and provides corresponding update rules as a parameter to AddAnimation, the latter can be reused as is.

Had we designed the operational semantics transformation to perform a single step only, it would have been possible to add animation using a multi-stage transformation approach as well (i.e., sequential execution of transformations). The transformation we discuss next, however, cannot be easily expressed by a multi-stage approach.

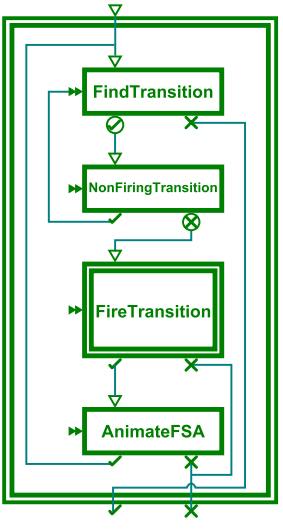

Fig. 11. Final Petri Net Control Flow

4.2 Correspondence Links

Animation update rules make use of correspondence links between finite state automata and Petri net models (see element "4" of Fig. 9). These correspondence links sometimes coincide with the intermediate generic links used in the "Finite State Automaton to Petri net" transformation, but the latter are not a reliable source for establishing correspondence. Furthermore, they are typically removed as part of the transformation in order not to waste memory space.

We can, however, automate the insertion of relevant correspondence links by adding correspondence associations between certain language elements at the metamodel level, i.e., by creating a "meta-triple" [15] (see Fig. 12). With this additional information, a higher-order transformation that extends the "Finite State Automaton to Petri net" transformation with the feature of establishing correspondence links, can be defined with a simple, single rule (see Fig. 13). This rule transforms translation rules such that they automatically insert corresponding links between respective input-output language element pairs. Note that Petri net places are linked to both state automaton states and events. It is therefore crucial to have the context of the original translation rule that creates an output language element based on the presence of an input language element. The rule in Fig. 13 specifically matches such creation

patterns. Establishing the correct links between corresponding input-output elements without this contextual knowledge would, in general, be impossible. This demonstrates that higher-order transformations cannot be subsumed by multi-stage transformations.

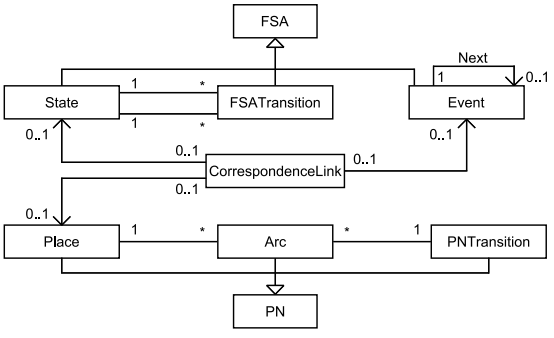

Fig. 12. Correspondence Meta Triple

Our correspondence higher-order transformation is not directly reusable because MoTif does not support parameterization of rules yet. Because of this limitation we could not formulate a generic version of the transformation that can be tailored to a particular application by passing in the names of metamodel elements of input/output language types. However, the transformation is reusable with respect to its structure. Another application can simply be obtained by a manual renaming of the input/output language types.

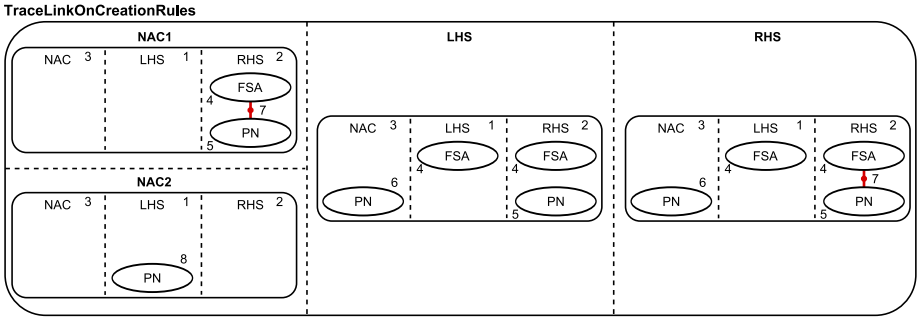

Fig. 13. H-O Transformation: Correspondence Links

5 Related Work

The need to relax conformance rules occurs in other areas as well. Levendovszky et al. capture domain-specific design patterns which also inherently are fragments of proper models [16]. Instead of creating a relaxed version of the metamodel, they use relaxed conformance, i.e., "relaxed instantiation". This allows them to use one original language definition to check both proper models and design patterns. Since they only need to support this one variant of conformance checking, this is a viable approach. In general, however, the explicit modeling of transformations may require a multitude of conformance levels, making the

relaxation of metamodels a more attractive option (see Sect. 3.2). Levendovsky et al., furthermore, observe that simply setting all minimal multiplicities to zero will allow the formulation of fragments which cannot be completed to proper models. They suggest detecting such fragments by using constraint solving. This approach is applicable in our context as well and could be realized by adding corresponding constraints to the relaxed metamodels.

Varró and Pataricza seem to have been the first to suggest higher-order transformations for improving the performance and maintainability of first-order transformations [17] (see Sect. 4).

Schürr's triple graph grammars are designed to support correspondence links between models from declarative rules [18]. In contrast to our respective higher-order transformation, a TGG transformation designer is more flexible in defining different correspondence links per rule. However, this also bears the risk that some correspondence links are forgotten or incorrectly established. A higher-order transformation like ours automates the process of establishing the required links and will be comparatively simple to correctly define, even for more advanced cases. Moreover, we may generate correspondence links with arbitrary amounts of additional information in contrast to the fixed format links of TGGs.

Jouault used ATL to define a higher-order transformation for automatically generating traceability links [19]. Unlike our correspondence link higher-order transformation, however, Jouault's TRACEADDER transformation adds *traceability* links between all elements rather then *correspondence* links. Traceability links can sometimes be used for tracking correspondences as well but not in general. Furthermore, while traceability links come for free as they do not need pattern specifications that only match relevant correspondences, they may use up a lot of memory albeit the majority of them is not being used in correspondence mapping applications. The main problem with using ATL for specifying higher-order transformations is that a higher-order ATL transformation has access to the transformation definition (e.g., to the FSA-to-PN transformation), but not to the latter's respective input and output languages (the FSA and PN metamodels). All matching patterns and output patterns for the input/output languages of the transformation are therefore unchecked as they occur on a purely textual basis only. This results in even mundane syntactic errors going unnoticed. This problem is aggravated by the fact that the result of the higher-order transformation must be tested dynamically in order to detect the errors. Sometimes the only means to detect errors is to examine the first-order transformation's output. Having detected errors in the output, one then has to trace back the errors to the first-order transformation and from there back to the higher-order transformation. While there will always be a class of errors that will require this extended backward reasoning, our approach can avoid this complicated procedure for purely syntactical errors.

6 Conclusion

Although we discussed our work and developed our artifacts in the context of ATOM^3/MOTIF, our ideas and results are by no means confined to the specifics

of this combination. Our proposal to explicitly model transformation definitions is applicable to a wide range of transformation approaches.

While it is not necessary to explicitly model *all* aspects of transformation definitions, we have illustrated that there are benefits associated with each such step. First, the explicit modeling of pattern specifications allowed the semi-automatic generation of customized pattern specification language definitions based on the components of relaxation, augmentation, and modification. It thus provided a cost-effective way to obtain customized transformation development environments. Second, the explicit modeling of transformation control structures allowed the modular addition of new behavior, such as source-level animation. Third, the explicit modeling of transformation rules allowed the automated enhancement of transformation rules, e.g., the insertion of correspondence links. With the latter application of a higher-order transformation we have demonstrated that such higher-order transformations cannot be replaced by a multi-stage approach using first-order transformations only.

This paper builds on our previous work [20] by demonstrating how our approach can facilitate higher-order transformations. The higher-order transformations we presented not only help to reduce the complexity of the base-transformations they are applied to, but are also reusable. The transformations we presented are furthermore applicable in a wide range of similar contexts. This and their reusability is a direct result of explicitly modeling all aspects of transformations including their control flow aspects. In contrast to ATL higher-order transformations, ours can be fully checked for well-formedness violations.

Summarizing, we provided further motivation for the utility of higher-order transformations, demonstrated the benefits of explicitly modeling transformations and proposed ways to economically enable their definition.

References

1. Atkinson, C., Kühne, T.: A tour of language customization concepts. Advances in Computers 70(3), 105–161 (2007)
2. 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)
3. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
4. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and metamodelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
5. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. SoSym 5(3), 261–288 (2006)
6. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
7. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation Specification (April 2008)

8. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Control flow support in metamodel-based model transformation frameworks. In: EUROCON 2005, Belgradem, Serbia, pp. 595–598. IEEE, Los Alamitos (2005)
9. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? transformation models! In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
10. Bézivin, J., Farcet, N., Jézéquel, J.M., Langlois, B., Pollet, D.: Reflective model driven engineering. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 175–189. Springer, Heidelberg (2003)
11. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009)
12. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with DEVS. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 136–151. Springer, Heidelberg (2008)
13. de Lara, J., Vangheluwe, H.: Automating the transformation-based analysis of visual languages. In: Formal Aspects of Computing, Special section on FASE (2008) (to appear)
14. Gorp, P.V., Keller, A., Janssens, D.: Transformation language integration based on profiles and higher-order transformations. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 208–226. Springer, Heidelberg (2009)
15. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. SoSym 6(6), 317–347 (2007)
16. Levendovszky, T., Lengyel, L., Mészáros, T.: Supporting domain-specific model patterns with metamodeling. Software and Systems Modeling, Theme Issue on Metamodeling (2009) (to appear)
17. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 290–304. Springer, Heidelberg (2004)
18. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
19. Jouault, F.: Loosely coupled traceability for atl. In: ECMDA Workshop on Traceability (2005)
20. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Systematic transformation development. ECEASST 21 (October 2009)