Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Synchronization of Abstract and Concrete Syntax in Domain-Specific Modeling Languages

By Mapping Models and Live Transformations

István Ráth[1], András Ökrös[2], Dániel Varró[1]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
[2] OptXware Research and Development LLC.
H-1137 Budapest, Katona J. u. 39., Budapest, Hungary

Received: date / Revised version: date

Abstract Modern domain-specific modeling (DSM) frameworks provide refined techniques for developing new languages based on the clear separation of conceptual elements of the language (called *abstract syntax*) and their graphical visual representation (called *concrete syntax*). This separation is usually achieved by recording traceability information between the abstract and concrete syntax using *mapping models*. However, state-of-the-art DSM frameworks impose severe restrictions on traceability links between elements of the abstract syntax and the concrete syntax.

In the current paper, we propose a mapping model which allows to define arbitrarily complex mappings between elements of the abstract and concrete syntax. Moreover, we demonstrate how live model transformations can complement mapping models in providing bidirectional synchronization and implicit traceability between models of the abstract and the concrete syntax. In addition, we introduce a novel architecture for domain-specific modeling environments which enables these concepts, and provide an overview of the tool support.

Key words domain-specific modeling languages – model synchronization – live model transformations – traceability

1 Introduction

Domain-specific modeling languages (DSMLs) play an increasingly important role in various areas of software engineering, including business process modeling or embedded systems. Their increasing popularity is due to the recent advances in domain-specific modeling (DSM) frameworks (like Eclipse Graphical Modeling Framework GMF [1]), which significantly accelerate the development of dedicated modeling editors with professional look-and-feel in a wide range of application domains.

Such DSM frameworks provide refined techniques for developing new languages based on the clear separation of conceptual elements of the language called *abstract syntax* (or *conceptual model*) and their visual representation called *concrete syntax* (or *diagram model*). This separation is achieved by precisely recording traceability information between the abstract and concrete syntax using so-called *mapping models* (or *trace models*).

Mapping models in the context of DSMLs can also be categorized from traditional traceability perspectives. For instance, first Eclipse-based DSMLs based upon the Graphical Editing Framework (GEF) used *internal traceability*, when concrete syntax elements had a direct reference to abstract syntax elements. More recent approaches like the Graphical Modeling Framework have opted for *external traceability* when the elements of the abstract and concrete syntax are interrelated via a separate mapping model.

However, unfortunately, even state-of-the-art DSM framework such as GMF impose severe restrictions on traceability links between elements of the abstract syntax and the diagram models. For instance, each concrete syntax element in a diagram must correspond to exactly one element in the underlying abstract syntax. Moreover, industrial DSM frameworks mostly provide automated synchronization between the abstract and concrete syntax in one direction: (i) changes of the diagram model initiated from the editor are propagated to the underlying abstract syntax, but (ii) direct changes of the abstract syntax frequently corrupt domain-specific editors (direct changes to the abstract syntax are usually due to editing actions in other concrete syntax representations, such as another diagram or a textual view, or may be results of automated model transformations). As a result, the use of DSM frameworks for developing complex, industrial strength visual modeling languages requires significant expertise and additional manual programming effort to overcome such difficulties.

The first claim for the current paper is that most of these problems are caused by the very simplistic handling of mapping models in DSM frameworks. For this purpose, we propose a *mapping model which allows to define arbitrarily complex mappings* between the abstract and concrete syntax of visual DSMLs. As a result, *visual abstraction* can be introduced to the graphical representation, i.e. a single graphical element may represent (abbreviate) a complex fragment of the underlying abstract syntax.

Obviously, such complex mapping models between the abstract syntax model and the diagram model introduces synchronization problems between the two models. Complex changes in the concrete syntax model (caused by editing operations) need to be immediately reflected in the underlying abstract syntax, and changes in the abstract syntax (e.g. caused by running background model transformations) would have a non-trivial effect on the concrete syntax model. For this purpose, mapping models will be processed with *incremental and live model transformations* [2,3], which continuously run in the background to immediately react to complex, non-atomic changes in (any of) these models in an incremental way. These reactions can be designed by relying on a high-level model transformation language [4].

Unsurprisingly, the handling of arbitrarily complex mappings between the abstract and concrete syntax have some architectural impact on the underlying DSM framework as well. Therefore, we demonstrate the practical feasibility of the approach using the ViatraDSM framework [5], which is a non-generative environment for developing DSMLs based upon the modeling and transformation features of the VIATRA2 model transformation tool.

The main contributions of the current paper are, therefore, the following: (1) we introduce a novel architecture for domain-specific modeling environments; (2) we propose a mapping model which allows to define arbitrarily complex mappings between the abstract syntax and the diagram model; (3) we demonstrate how live transformations can support to maintain the coherence of these models, (4) we provide (an overview of the) tool support. These concepts will be demonstrated by developing a simple still representative domain-specific modeling language.

Taking a traceability viewpoint, our proposal combines *explicit traceability* (the traditional way when traceability links between two models are explicitly persisted to a mapping model) with *implicit traceability* (when traceability is provided implicitly by a live model transformation transformation running in the background as a daemon). In fact, it is a design decision (depending on the application domain and traceability requirements) how to balance between the two approaches. In the current paper, we use generic, declarative mapping models for explicit traceability (which can be reused for capturing traceability between other modeling languages). Then one-to-one mappings between source and target elements are handled by generic model transformation rules. However, for more complex (arbitrary m-to-n) synchronizations, we use designated (domain-specific) transformation rules.

The rest of the paper is structured as follows. Sec. 2 summarizes main concepts of developing domain-specific modeling languages, and using their corresponding editors. To better motivate our work, we give an overview of the state-of-the-art Eclipse Graphical Modeling Framework (Sec. 2.3), and identify its architectural problems and limitations. Sec. 3.1.1 presents a novel architecture (exploited in the ViatraDSM framework) for the solving the synchronization problems in DSMLs. Corresponding models (abstract syntax, diagram, mapping) are introduced in Sec. 3.1.3–3.1.5 exemplified on Petri nets as a DSML (Sec. 3.1.1). Sec. 3.2 presents live model transformations as provided by the VIATRA2 model transformation framework. Then, Sec. 4 presents our solution for the generic synchronization of the abstract and concrete syntax of DSMLs using mapping models and live transformations. Sec. 4.4 provides a brief overview on implementation details. Related work is assessed in Sec. 5, and finally, Sec. 6 concludes our paper by highlighting additional applications of our mapping model and synchronization techniques.

2 Challenges of model synchronization in graphical editors

In this section, we use the Eclipse Graphical Modeling Framework as a modern, state-of-the-art domain-specific language engineering environment as the problem context. It is important to note that the ideas and issues explained are not specific to GMF, in fact, they represent a generalization of experience gathered in designing and implementing custom domain-specific languages with various technologies.

2.1 Design of Domain-specific modeling languages

In domain-specific visual language design, the three most important design aspects are the following:

– *Abstract syntax* specification, which is typically carried out using metamodeling. The basic notions of the language (model elements) and their relations (associations) are defined in a mathematically precise way, with structural constraints (e.g. to express containment relations, or type correctness for associations), multiplicities and implicit relationships (such as inheritance, refinement).

– *Concrete syntax* specification targets the actual visual appearance of the language, assigning a visual symbol to those language elements which are to be represented on diagrams.
– *Language constraints* are frequently also needed, to express correctness criteria that are cannot be specified using metamodeling (e.g. attribute value validity intervals, or complex structural well-formedness rules that involve multiple model element configurations).

Early domain-specific modeling tools such as MetaCase's MetaEdit+ [6] derive the *structure* of the graphical representation from the abstract syntax, as notation definitions are assigned directly for each abstract syntax model element. This is suitable for simple languages with a few element types, however, in today's practical applications, language metamodels are becoming increasingly large and complex. As a result of the mapping approach, the complexity is propagated into visual diagrams.

Tackling this visual complexity is a major challenge in designing domain-specific modeling languages on the right level of abstraction, which simultaneously provides (i) intuitive graphical syntax without unnecessary details, and (ii) an abstract syntax close to the concepts of the domain.

2.2 Architectural overview of DSMs

A straightforward strategy to balance abstraction with expressive power is to separate abstract and concrete syntax representations. Essentially, this approach treats the visual notation as a separate language with its own element types, attributes and relations, on an additional modeling layer. For two dimensional graph-like languages (as most visual languages are conceptualized), this visualisation grammar is derived from a *core diagram metamodel*, which contains attributed *nodes* and *edges*. By refining these concepts to specific model elements, the structure of the concrete syntax may be elaborated; visual appearance is specified by traditional design tools as previously.

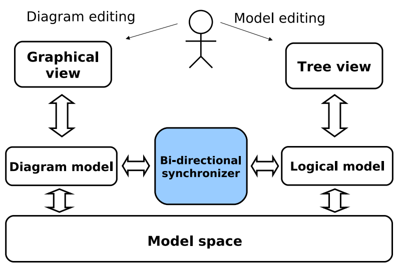


Fig. 1 Conceptual overview of domain-specific editors

In graphical DSMs, the model is typically manipulated by the designer using a graphical editor over the concrete syntax. Changes initiated in the concrete syntax are immediately propagated to the abstract syntax model. However, in many DSMs, there are certain model elements, which are not visible to the user in the concrete syntax, thus they need to be manipulated directly in the abstract syntax (e.g. by using a Property sheet). Moreover, modern DSM may offer multiple visualizations (diagrams, textual notation, hierarchical overviews) of the same abstract syntax model. In this case, a change in one concrete syntax triggers a change in the abstract syntax, which needs to be reflected instantaneously in the other visualizations. Finally, many complex model manipulations (for model analysis or model transformations) are carried out directly on the abstract syntax, and the result of their execution needs to be reflected in all concrete syntax views preferably immediately. All these scenarios highlight that bidirectional synchronization of various models of the abstract and concrete syntax is a major challenge for DSMs.

For this purpose, a modeling environment typically offers a *hybrid view* of the model space. Since the user is working with two separate notations of the same model, synchronization has to be done on-the-fly. As abstract and concrete syntax models are stored in separate modeling layers, the solution is a model-to-model synchronizer which maintains both representations and maps changes symmetrically.

2.3 The GMF approach and its limitations

As a state-of-the-art environment, the Eclipse Graphical Modeling Framework follows this design pattern.

At run-time, GMF maintains two distinct model instances: the abstract syntax models conforming to the ECore metamodel, and a *Notation model* conforming to a built-in Notation metamodel. Notation models correspond to diagrams and contain only visualisation-specific information. The user is interacting with a "parameterized", but generic Notation model editor, where these parameters contain information on how diagrams can be manipulated and mapped to domain models.

GMF automatically performs the mapping as the user is editing the model, according to a built-in semantics. Model synchronization is implemented by using a simple traceability mechanism, where each Notation model element references a corresponding abstract syntax counterpart. This trace model is contained in the Notation model resource, and implements a simple *one-to-one* mapping (with the exception of labels, which may reference several attributes through special format strings). By this approach, GMF is able to partially separate abstract and concrete syntax representations.

Unfortunately, in case of advanced applications, severe problems arise due to architecture-level design decisions and limitations of GMF.

1. The GMF semantics is *restricted to one-to-one mappings* between model elements of the abstract syntax and their graphical representation. In case of complex modeling languages (like AUTOSAR), abstraction capabilities of the concrete syntax would be advantageous e.g. to allow a graphical notation to abbreviate more than a single element in the underlying concrete syntax.
2. *It is impossible in GMF to fully separate the abstract syntax and the concrete syntax of a language.* In fact, GMF imposes some (hidden) "meta-constraints" on the abstract syntax such as the existence of a Diagram notion in the metamodel, or the connection classes providing navigation from both directions. Additionally, as the abstract syntax model is not independent from GMF's visualisation, significant development efforts are required to tailor existing Eclipse Modeling Framework (EMF [7]) models to be GMF-compliant.

As the authors themselves experienced in real development projects in the automotive domain, overcoming these problems of frequently require significant programming effort specific to the modeling language itself. For instance, in case of the industry-standard AUTOSAR metamodel, developing a GMF based editor requires the creation of a new GMF-compliant metamodel (for the sublanguage which is planned to be visualized), with expensive ad hoc synchronizers in-between (as illustrated in Fig. 2).

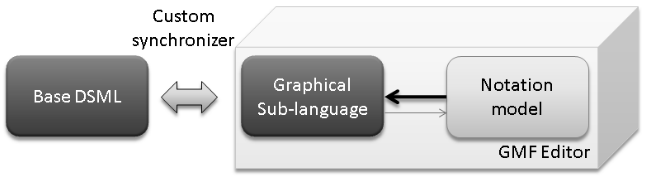


Fig. 2 Implementation pattern

3 Preliminaries

3.1 Modeling the Abstract and Concrete Syntax

3.1.1 Novel architecture for synchronizing abstract and concrete syntax In the current paper, we propose a novel solution to completely separate the abstract syntax and concrete syntax of a graphical modeling language with arbitrary mapping between them using advanced traceability models and live model transformations. Our approach will be demonstrated using ViatraDSM, which is a framework for developing domain-specific modeling languages using a novel underlying architecture (depicted in Fig. 3). The essence of the solution can be summarized as follows.

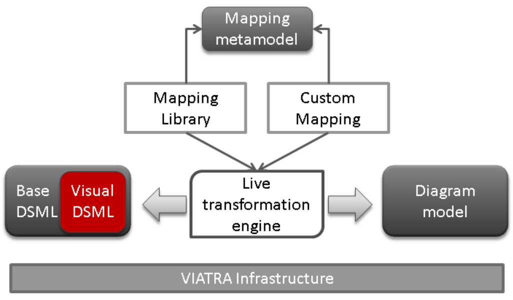


Fig. 3 The ViatraDSM architecture

– A *general mapping model* is used to connect elements of abstract and concrete syntax, which significantly extends the capabilities of GMF mappings.
– *Metamodel-tagging* is used for the abstract syntax (conceptual metamodel) of the modeling language, which eliminates need for introducing a separate conceptual sub-language (as frequently necessitated by GMF).
– A *general model transformation language* is used to specify more complicated mappings between the abstract and concrete syntax on a high-level of abstraction (namely, as a model transformation solution instead of pure Java code).
– *Live model transformations* are used as an execution mechanism driven by changes in the underlying models to achieve high performance even for large models.
– A *mapping library* is provided as a guideline to accelerate the implementation of abstract-concrete syntax mappings.

Petri nets as a domain-specific language. Throughout the paper, we will use the domain-specific language of Petri nets as a demonstration case study. We use Petri nets extended with *arc weights* and *place capacities*. Arc weights are integers associated to both Input Arcs and Output Arcs, and determine the amount of tokens that a given arc must carry. Place capacities simply impose a restriction on the amount of tokens assigned to a place at any given time.

3.1.2 A synchronization problem between abstract and concrete syntax As a demonstrating example, we will construct an advanced domain-specific modeling environment for the Petri net language (using the architectural considerations discussed in Sec. 2.2 and 3.1.1). The editor provides a graphical concrete syntax representation for Petri net graphs, with support creating *Place* and *Transition* nodes and *In/OutArc* edges (ensuring syntactic correctness while editing, i.e. an *OutArc* can only start at a *Place*). Graphical attributes (such as *token count* for Places and *arc weight* for Arcs) can be edited through a standard property editor. As a framework for the example, the ViatraDSM domain-specific environment [5] based on the VIATRA2 model transformation system will be used.

Our example editor implements the abstraction shown in Fig. 4. It involves "hiding" token instances and displaying the number of associated *Tokens* as a numeric label inside the *Place* instance, whereby a derived model property (the number of token instances) is mapped to a diagram element property (tokenCount).

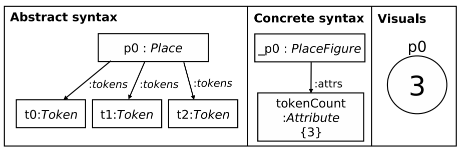
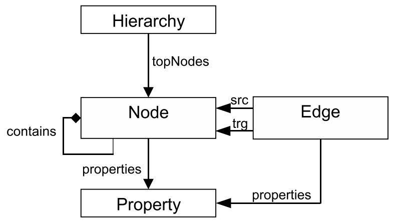


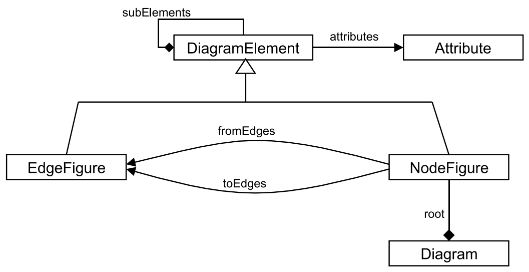
Fig. 4 Model layers in the Petri net editor

3.1.3 Metamodels for abstract and concrete syntax In the following sections, we will provide a precise metamodel-based approach to support arbitrary abstract-concrete syntax mapping abstractions.

Defining abstract syntax. The standard workflow of creating an editor in the domain-specific framework begins by creating the abstract syntax metamodel.



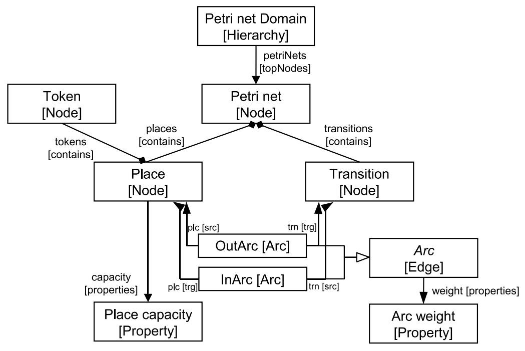
(a) Core abstract syntax metamodel



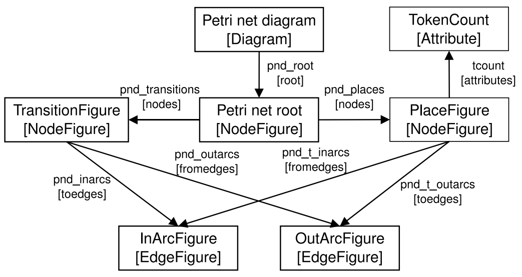
(b) Core concrete syntax metamodel

Fig. 5 ViatraDSM's core metamodels

Figure 5(a) shows the core metamodel for abstract syntax (*domain*) model elements associated to a *Hierarchy*. It defines a directed, labeled graph with *Nodes* and *Edges*; both Nodes and Edges can have *Properties*. Nodes are organised into a *containment* hierarchy. In ViatraDSM, language metamodels are *refinements* of the core metamodel. This refinement relation is established by subtyping, effectively "tagging" the elements of the domain metamodel with elements from the core metamodel. By this approach, any domain-specific language can be mapped making it easy to separate a visual sublanguage of the base DSML according to Fig. 3. For instance, in the case of the Petri net domain metamodel (Fig. 6(a)), a *Token* instance is only allowed to be added to a *Place* instance because of the *tokens* relation which is a refinement of the core *contains* concept.



(a) Abstract syntax metamodel



(b) Concrete syntax metamodel

Fig. 6 Petri net metamodels

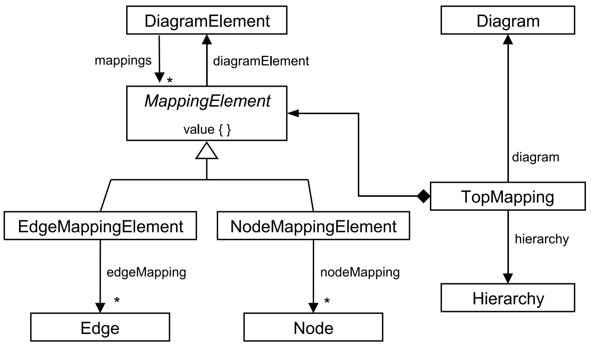
Defining concrete syntax. Similarly to the abstract syntax metamodel, a concrete syntax (*diagram*) metamodel may be defined. For the Petri net editor example, Figure 6(b) shows a simple diagram metamodel as a refinement of the core concrete syntax (diagram) metamodel (Fig. 5(b)). In this case, *Petri nets*, *Places* and *Transitions* are mapped to separate visual graph nodes (*Petri net root*, *PlaceFigure* and *TransitionFigure*, respectively), while *OutArcs* and *InArcs* are visualised as graph edges (*OutArcFigure* and *InArcFigure*).

Note that *Tokens* are not included in the diagram metamodel. Instead of mapping them to separate visual nodes, this visualisation language uses an attribute (*TokenCount*) to indicate the number of tokens assigned to a place. This approach corresponds to the complete abstract-syntax separation principle laid out in Sec. 3.1.1 and provides the basis for solving the problem in Sec. 3.1.2.

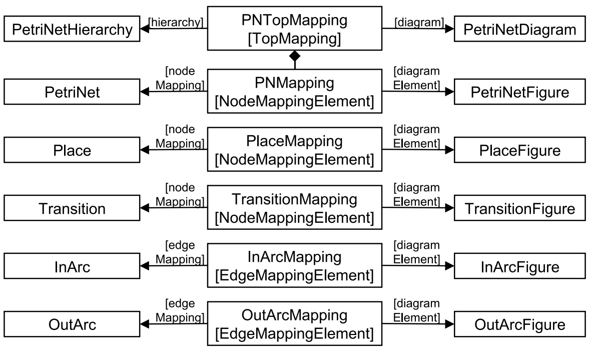
3.1.4 Trace metamodels between abstract and concrete syntax. To maintain a consistent mapping between instances of the abstract and concrete syntax metamodels, we propose to use *trace models* conforming to a generic trace metamodel.

For trace models, we use the core metamodel in Fig. 7(a). It defines a containment hierarchy between *TopMapping*s and *MappingElement*s. A *TopMapping* instance connects a concrete syntax *Diagram* with an abstract syntax *Hierarchy*, so it serves as a top-level container for the rest of the trace metamodel elements. Each *DiagramElement* (*NodeFigures*, *EdgeFigures*) may have multiple *MappingElement* associations; since *MappingElement* is abstract, two non-abstract subclasses (*EdgeMappingElement* and *NodeMappingElement*) are used to define trace bindings to abstract syntax *Edges* and *Nodes*, respectively. This way, a flexible *n-to-m* (many-to-many) mapping can be defined, where a concrete syntax element may reference multiple abstract syntax elements by defining multiple *MappingElements*, or, an abstract syntax element may be connected to multiple *MappingElements* in the reverse direction.

It is important to note that *MappingElements* have a *value* attribute, which may be used to store values of any kind; this is important since in this way, the trace models are capable of persisting non-structural state information (e.g. by storing "old" values of a model attribute).



(a) The core mapping metamodel

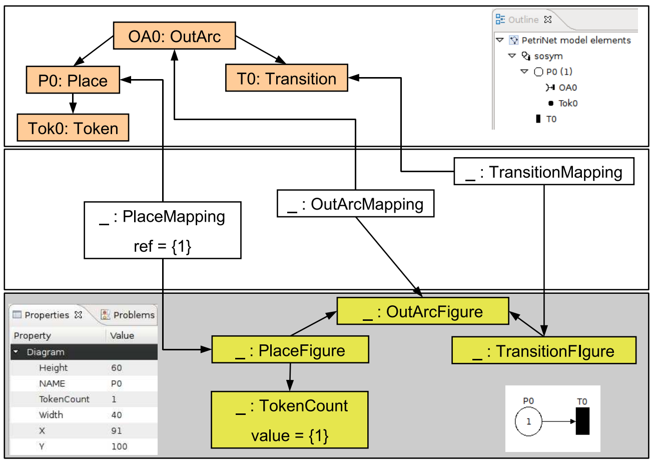


(b) Example of a mapping metamodel (Petri net domain)

Fig. 7 The mapping metamodel and its usage for the Petri net domain

The refinement of the core metamodel to the Petri net diagram domain is shown in Fig. 7(b). In this case, we have constructed a partial *one-to-one* mapping between the metamodels in Fig. 6(a) and 6(b). Note this is still a *metamodel*, thus it represents a domain-specific variant of the core mapping concepts; also, certain details, e.g. containment relations between *PNTopMapping* and *NodeMappingElements* have been omitted from Fig. 7(b) for the sake of retaining visual clarity.

3.1.5 Trace model instances. Fig. 8 demonstrates how abstract, concrete, and trace model instances may be interconnected in an actual modeling scenario. In this case, the Petri net consists of a place `P0` with a token `Tok0`, connected by an outarc `OA0` to a transition `T0`. This abstract syntax model, as shown with orange, is presented to the user by a tree view (top right). On the diagrams, a concrete syntax model (yellow) is rendered where the token count is shown both graphically (bottom right) and in the properties view (bottom left). Not directly visible to the user, the system maintains the trace model (shown in white), which encodes the logical mapping between the two model representations.



Property	Value
Diagram	
Height	60
NAME	P0
TokenCount	1
Width	40
X	91
Y	100

Fig. 8 Model instances

The *PlaceMapping* instance stores a reference value of 1, which is used to store the mapped value of the *TokenCount* attribute. As the user changes this attribute value by entering the new value in the Properties view, the system reacts automatically and adjusts the number of token instances assigned to P0 in the abstract model (concrete → abstract synchronization). Symmetrically, should the user perform editing directly in the abstract syntax model (through the tree view), e.g. by adding another token to P0, the system keeps track of this change and applies the necessary modification to the graphical view by updating the tokencount attribute value.

3.2 Live Model Transformations

By our approach, bi-directional synchronization is accomplished by live model transformations (introduced in [3]), which incrementally react to various changes of models including *atomic* model updates as well as a complex *sequence* (set, transaction) of such atomic operations. Prior to the technicalities of live transformations, we first give an overview of the related fragment of the VIATRA2 model transformation language [4]. Then live transformations will be intensively used as means of incremental synchronization between the abstract and concrete syntax models of DSMLs later in Sec. 4.

3.2.1 Model transformation language overview. The VIATRA2 Framework's transformation language consists of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) [8] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [9] rules can be used for the description of control structures.

Graph patterns are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. The basic pattern body contains model element and relationship definitions. In VIATRA2, *patterns may call other patterns* using the *find* keyword. This feature enables the reuse of existing patterns as a part of a compound pattern. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled. A *negative application condition* (NAC, defined by a negative subpattern following the *neg* keyword) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations).

Graph transformation (GT) [8] provides a high-level rule and pattern-based manipulation language for graph models. In VIATRA2, graph transformation rules may be specified by using a *precondition* (or left-hand side – LHS) pattern determining the applicability of the rule, and a *postcondition* pattern (or right-hand side – RHS) which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged. Further actions can be initiated by calling any ASM instructions within the *action* part of a GT rule, e.g. to report debug information or to generate code. In addition to graph transformation rules, VIATRA2 provides procedural constructs (such as simple model operations – new, delete, update) as well as pattern and scalar variables. Using these constructs, *complex model transformations* can be written.

3.2.2 Overview of the live transformation approach.

Match set. In our approach, a model change is detected by a change in the *match set* of a graph pattern. The match set is defined by the subset of model elements satisfying structural and type constraints described by the pattern. Formally: a subgraph S of the model G is an element of the match set M(P) of pattern P, if S is an isomorphic image of P.

Detecting model changes. Changes in the match set can be tracked using the RETE network [10]. A model change occurs if the match set is expanded by a new match or a previously existing match is lost. Since a graph pattern may contain multiple elements, a change affecting any one of them may result in a change in the match set. The RETE-based incremental pattern matcher keeps track of every constraint prescribed by a pattern, thus it is possible to determine the set of constraints causing a change in the match set. Our approach can be regarded as an extension of the *fact change* approach [11]. It provides support for the detection of changes of arbitrary complexity; not only atomic and compound model change facts (with simple and complex patterns respectively), but also operations, or sequences of operations can be tracked using this technique (either by representing operations directly in the model graph, or by using reference models).

Triggers. Our approach is intended to support a broad range of live transformations. For this purpose, incremental transformation rules, called *triggers* are explicitly specified by the transformation designer. The formal representation of a trigger is based on a simplified version of the graph transformation rule: it consists of a *precondition pattern* and an *action* part consisting of a sequence of VIATRA2 transformation steps (including simple model manipulations as well as the invocation of complex transformations).

```
@Trigger(sensitivity='rise')
gtrule newPlace() = {
 precondition pattern place(P) = {
  Place(P);
 }
 action {
       print("A place appeared: "+name(P));
}}
```

Listing 1 A simple graph transformation trigger

In Listing 1, a simple trigger is shown. It is automatically fired after the user creates a new Place instance, since the trigger is activated for a newly found match (`sensitivity=rise`). We use the `Trigger` annotation as an extension to the VIATRA2 language to indicate that the graph transformation rule should be executed as an event-driven transformation. The *sensitivity* annotation can take two other parameters (*fall* and *both*) – *fall* triggers are executed when a previously existing match is lost; *both* triggers execute on rises and falls as well.

Execution context. The system tracks changes changes in the match sets of patterns and executes the action sequences in a persistently maintained *execution context*. This context consists of *pattern variables* (continuously maintained by the RETE network[1]) and *persistent variables* (called *ASM functions* in VIATRA2; essentially global associative arrays).

3.2.3 Complex change detection in triggers. To detect complex model changes, the transformation developer can primarily make use of the *rise* and *fall* triggers (and some advanced VIATRA2 pattern language constructs).

Creation. In practical applications, a chain of triggers may be used to execute multiple incremental updates. For instance, after a Token instance has been added by the user, the system may execute a trigger which automatically connects it to a Place (Listing 2, *tokenAdded*).

```
@Trigger(sensitivity='rise')
gtrule tokenAdded() =
{
 precondition pattern token(T) = {
   Token(T);
 }
 action {
   println("A token was added: " + name(T));
   // action: find a place and connect the
   // unconnected token to it
   choose P with find place(P) do
    new Place.tokens(_,P,T);
}}

asmfunction numberOfTokens / 1;

@Trigger(sensitivity='rise',priority=1)
gtrule tokenConnected() = {
 precondition pattern connectedToken(P,T) = {
  Place(P);
  Token(T);
  Place.tokens(_,P,T);
 }
 action {
   update numberOfTokens(P) = numberOfTokens(P) + 1;
}}
```

Listing 2 Trigger to handle the addition of Tokens

After *tokenAdded* has fired, another trigger similar to Listing 1 (*tokenConnected*) updates the *numberOfTokens* array stored in the execution context.

Deletions. To detect deletions, a trigger for the same precondition pattern as used in Listing 2 can be used in *fall* mode. In this case, a *when-clause* is used to filter the case when the match set loss occured because of the deletion of a model element referenced by the T variable (Listing 3). Other pattern variables (pointing to existing model elements) can be used in the action part in the usual way.

```
@Trigger(sensitivity='fall',priority=1)
gtrule tokenRemoved() = {
 precondition find tokenAdded.connectedToken(P,T)
 action {
  // only act if token T has been lost (deleted)
  when (delete(T)) seq {
   update numberOfTokens(P) = numberOfTokens(P) - 1;
}}}
```

Listing 3 Handling token deletion

Attribute updates. The system also provides support for the incremental detection of attribute changes. VIATRA2 provides a *value* field for all node types; in this example, this value field of the *Place capacity* property node is used to store the actual value of the capacity of the connected *Place* (see Fig. 6(a)).

```
// associative array to cache place capacity values
asmfunction capacities / 1;

@Trigger(sensitivity='fall')
gtrule capacityChanged() = {
  precondition pattern pre(P,PC) = {
    Place(P);
    'Place capacity'(PC);
    Place.capacity(_,P,PC);
    // check condition to define a value constraint
    check(value(PC) == capacities(PC))
  }
  action {
    // check whether the attribute update
    // caused the activation
    when (update(value(PC))) seq {
     // update the cache
     update capacities(PC) = value(PC);
     // the user has changed the attribute
     // check if the constraint effectively holds
     if (!(numberOfTokens(P) =< value(PC))) {
      // constraint is violated, notify the user
      println("The capacity constraint "+
                "is violated at: "+name(P));
}}}}
```

Listing 4 Handling attribute updates

In Listing 4, a *fall* trigger is defined for changes in the capacity value (the user may change that any time during modeling). The trigger is activated for changes in the match set of a complex pattern involving a *match check condition*, which is a special feature of the VIATRA2 transformation language to define additional attribute constraints which cannot be expressed using structural graph patterns. The global array *capacities* is used to cache known capacity values; the trigger checks whether the cause of activation was a change in the attribute value and proceeds to update the cache and notify the user if the validity of the capacity constraint is violated in the given context.

4 Generic abstract-concrete synchronization with mapping models and live transformations

In this section, we combine our live transformation approach (as presented in Sec. 3.2) with the trace meta-

[1] This means that the matches stored in a given pattern variable are always updated and the match set of any pattern can be retrieved in constant time.

models (Sec. 3.1.3) to provide a generic, metamodel-driven transformation approach for the on-the-fly synchronization and tracing of abstract and concrete syntax representations of a graphical domain-specific language. By our approach, the two modeling layers can be fully separated, making arbitrary visualisation abstractions possible.

First, we describe the core cases of simple *one-to-one* model synchronizations (Sec. 4.1). These *synchronization primitives* build on reference mapping models to define graph transformation rules for handling model creation, deletion and attribute updates.

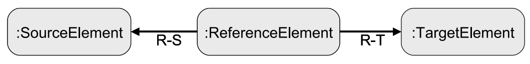
Based on these primitives, we propose a *Mapping Library* (Sec. 4.2), which uses the metamodels introduced in Sec. 3.1.3–3.1.5. Our generic approach is applicable to any domain, as it follows correspondence pairs encoded in a *mapping metamodel.* This library replicates GMF's mapping functionality for ViatraDSM editors, but also provides a flexible starting point for further mapping customization in order to allow language engineers to go beyond GMF's mapping capabilities.

Finally, in Sec. 4.3, we demonstrate that by combining the basic techniques from Sec. 4.1 with the generic approach in Sec. 4.2, a language engineer may create custom mappings easily. As a proof-of-concept, we solve the abstraction mapping problem outlined in Sec. 3.1.5.

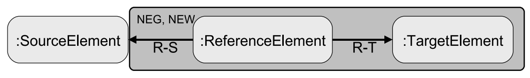
4.1 Trace models in live transformations

In a synchronization scenario, a source and a target model (or an entire source graph and target graph) are present, which have to be kept synchronised at all times. Note that usually this is not merely a batch model transformation from a source model to new target models, both models evolve concurrently. Every modification on the source model has to be followed on-the-fly with the relevant modification on the target model. Furthermore, the consistency of the models has to be maintained, so every change in the target model which is relevant to the source model, has to be handled as well. Usually these two models are fully separated from each other. This means that a model element in the source model may not have a direct link (reference) to its corresponding model element in the target model. Moreover, multiple elements from the source model can be related to a single target element, hence it is necessary to use trace models which connect source and target models.

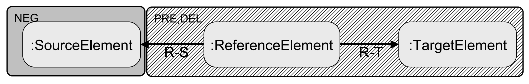
In the following general example, we outline the core cases of the *source* → *target* synchronization scenario. In Fig. 9(a), a consistent state of the model space is shown, where a *SourceElement* instance is connected to a *TargetElement* instance by a trace model instance of type *ReferenceElement.* This configuration expresses the correspondence relationship between the source and target model elements.



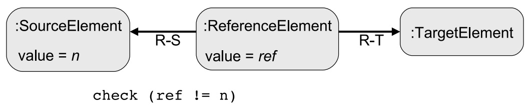
(a) A consistent state of the model space



(b) Source element creation



(c) Source element deletion



(d) Attribute value change

Fig. 9 Basic patterns of using trace models for live synchronization

4.1.1 Detecting element creation in the source models.

In the following scenario, we define a live transformation fragment which detects that a new source model element has been created and creates the corresponding target model element (Fig. 9(b)). The NEG area marks a *negative application condition* over the reference and target model elements. Therefore, a *rise* trigger will fire when a new source model element which does not yet have a corresponding pair in the target model, has been created. The action sequence of the graph trigger will then proceed to create exactly those elements which are included in the negative application condition (as indicated in Fig. 9(b) by the NEW keyword).

4.1.2 Detecting element deletion in the source models.

As long as the model space is kept consistent, every *SourceElement* has a *TargetElement* pair, and an appropriate *ReferenceElement* with its relations. As a consequence, deletions in the source model hierarchy can be handled by a graph transformation trigger shown in Fig. 9(c). After deleting a *SourceElement*, a *ReferenceElement* remains, without the *R-S* relation.

We again use a negative application condition (marked NEG) in a rise trigger to detect a new occurrence of a such a *ReferenceElement–TargetElement* stub (PRE indicates that both the *ReferenceElement* and *TargetElement* instances are included in the precondition). In the action sequence, the graph transformation rule will proceed to delete the *ReferenceElement* and the *TargetElement* instances.

4.1.3 Detecting attribute updates in the source models.

While attribute value changes can be detected using techniques described in Sec. 3.2.3, detecting a change with respect to the *last synchronized value* involves storing values in the trace models. As shown in Fig. 9(d), we define an *attribute check condition* on the value equality of attributes stored in the source and trace models.

In this way, the pattern matcher will detect when an attribute update has occurred. Note that the action sequence is omitted from Fig. 9(d) since it may be domain-specific (e.g. the trigger may fire an attribute update in the target models).

4.2 Generic abstract–concrete syntax mapping

By combining the basic techniques described in Sec. 4.1 with trace metamodeling and modeling as shown in Sec. 3.1.3 and 3.1.5, we show a generic approach to abstract-concrete synchronization. Conceptually, this approach establishes a *Mapping Library* to provide a GMF-like one-to-one mapping facility. Note that due to space constraints, we focus on illustrating the core ideas; the complete implementation is available as part of the standard VIATRA2 software distribution.

A metamodel-driven generic transformation takes a specification metamodel as an input to determine rules that describe how the transformation should be performed. In the trace metamodel (Fig. 7(b)), corresponding source and target model element *types* are connected with *MappingElement* types to indicate, for instance, that any given *Place* instance should be mapped to a *PlaceFigure* instance and vice-versa. Note that the core mapping metamodel (Fig. 7(a) and 5(a)) allows assigning multiple abstract syntax elements to a concrete syntax element: for instance, (i) multiple *Nodes* and *Edges* may be assigned to a *MappingElement*, and (ii) multiple *MappingElements* may be assigned to a *DiagramElement*) supporting a *many-to-many* mapping semantics. In this example, we demonstrate a more simple, *one-to-one* correspondence which is analogous to GMF's capabilities.

The graph transformation triggers below are presented in a compacted notation. In the figures, abstract syntax model elements appear on the left (with the `AS_` prefix for pattern variables), while concrete syntax elements appear on the right (`CS_`).

4.2.1 Capturing types in graph patterns. Fig. 10 shows the generic *existsInMetaModel* subpattern which demonstrates how graph triggers can be defined to match *any domain*. This subpattern matches domain metamodel elements (subtypes of the core *NodeFigure*, *NodeMapping* and *Node* elements; subtyping is denoted shortly by square brackets) and provides pattern variables (`CS_TYPE, TR_TYPE, AS_TYPE` in Listing 5) which pass type information regarding contextual information captured in the mapping metamodel. For instance, these pattern variables may take model references to *PlaceFigure*, *PlaceMapping*, and *Place* as values.

```
pattern existsInMetaModel(AS_TYPE, CS_TYPE, TR_TYPE)=
{
 // refinement of core domain metamodel
 supertypeOf(MetaNode, AS_TYPE);

 // refinement of core mapping metamodel
 supertypeOf(MetaNodeMappingElement, TR_TYPE);

 // connecting relationships
 relation(_,TR_TYPE,AS_TYPE);
 relation(_,TR_TYPE,CS_TYPE);
}
```

Listing 5 VIATRA2 code for the `existsInMetaModel` pattern

4.2.2 Tracking creation in the concrete syntax. In Fig. 11(a), the *linkNodeFigure* trigger is presented. This trigger creates domain-specific *Nodes* for every *NodeFigure* which is created by the user during model editing (the direct type of Nodes and NodeFigures is passed as pattern variables from Fig. 10). Since the concrete syntax metamodel allows for creating concrete syntax nodes in two contexts (as *top nodes* placed directly on the diagram and as *sub nodes* of a container node), the live transformation sequence has two modes of operation. Correspondingly, the precondition pattern of the *linkNodeFigure* trigger is an *OR* pattern, which defines a logical disjunction for each of the cases[2].

Both subpatterns share the same structure; a negative application condition (marked with dark grey) ensures to match against concrete syntax model elements, which do not yet have corresponding mapping and abstract syntax elements. Note that *NodeFigure, Diagram, Hierarchy, TopMapping, NodeMapping* are *indirect*, generic types in this case, the direct domain-specific type is only relevant for the concrete, mapping and abstract syntax nodes (`CS_NODE, TR_NODE_MAP, AS_NODE` are tagged with type values `CS_TYPE, TR_TYPE, AS_TYPE` respectively).

The trigger creates these missing elements, both the abstract syntax node and the mapping node with connecting relationships, similarly to the *creation synchronization primitive* in Sec. 4.1.1.

4.2.3 Tracing deletions. Fig 11(b) shows the *deleteHandling* trigger, which demonstrates how to detect deletion in both abstract and concrete syntax models.

This *rise* trigger also references the generic subpattern in Fig. 10 for type information. We use a disjuntive OR-pattern to handle the following cases:

– The first OR-subpattern corresponds to the case where a concrete syntax node has been deleted. This is signaled by the appearance an abstract syntax node with its related mapping element without a related concrete syntax node. As a reaction, the mapping model element has to be deleted along with the related abstract syntax element, in parallel with the GMF mapping semantics ("delete from model" operation).

[2] OR patterns are matched if any of the disjunct subpatterns match.

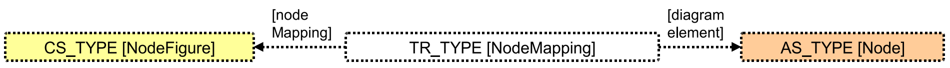
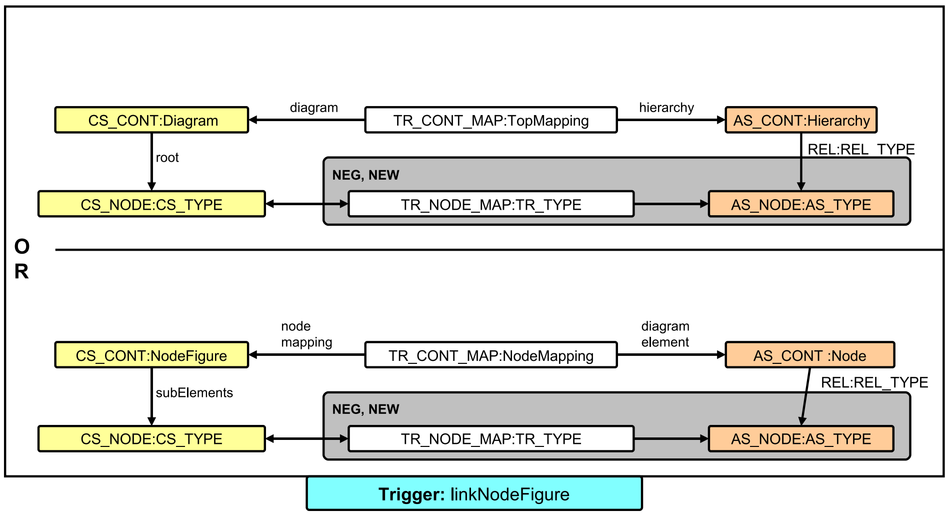
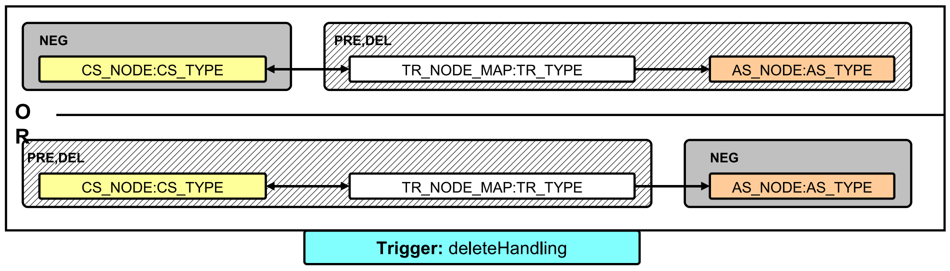


Fig. 10 The *existsInMetaModel* generic graph pattern



(a) The linkNodeFigure graph trigger



(b) The deleteHandling trigger

Fig. 11 Model synchronization triggers

– In the second case, we define an OR-subpattern which corresponds to the case when an abstract syntax element is deleted. This event is signaled by the appearance of a concrete syntax node – mapping model node pair without a connected abstract syntax node. As a reaction, the concrete syntax element is be deleted from the graphical representation along with the mapping element. Note that extending the basic techniques to bidirectional synchronization is straightforward, since symmetries can be easily exploited in pattern definitions.

4.2.4 Relevance In Sec. 4.2, we have highlighted the foundations of a generic *Mapping Library*, which leverages our mapping metamodel and live transformation technology to provide a general solution for the incremental synchronization of abstract and concrete syntax models. By combining simple techniques, our approach provides a GMF-like one-to-one mapping semantics, which works for arbitrary domains and is specified using a high abstraction level transformation language.

4.3 Arbitrary abstract–concrete syntax mapping

In practical applications, the need for a custom mapping frequently arises, where a mapping rule framework is needed for a special abstraction, e.g. to simplify the visualisation of a complex modeling language. In state-of-the-art frameworks such as GMF, the language engineer is stuck with the default options provided, and customization beyond those requires extensive programming, which is only possible when the application programming interface allows for a straightforward programmatic hook at the right places.

In contrast, our transformation-driven approach provides extensibility and customization at a significantly higher level of abstraction. By simply defining custom graph triggers or overriding the ones provided by the Mapping Library, the language engineer may customize abstract-concrete syntax mappings using the techniques shown previously. In the following example, we use the context of the Petri net case study (Sec. 3.1) to demonstrate a common mapping abstraction, where the visualisation layer presents the number of model elements of a certain type (tokens) as a simple numeric attribute, instead of assigning a graphical diagram element to each (Fig. 4).

This custom synchronization transformation has to perform two tasks:

– *concrete → abstract syntax synchronization*: whenever the attribute value in the concrete syntax model changes (e.g. the user changes its value through the GUI), the appropriate number of *Token* instances should be assigned to the *Place* instance the owner of the changed attribute was mapped to (by creating new tokens or deleting existing ones).
– *abstract → concrete syntax synchronization*: symmetrically, when a new *Token* instance is assigned to the *Place*, or a previously existing one is deleted, the attribute value of the *PlaceFigure* must be updated accordingly.

4.3.1 Tracking the concrete syntax model. We use the technique shown in Sec. 4.1 to define a precondition pattern for detecting changes in attribute values in the concrete syntax model (Fig. 12).

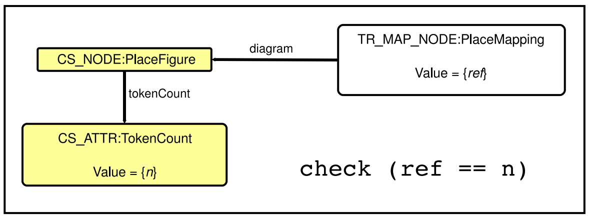


Fig. 12 Tracing attribute value changes in the concrete syntax

4.3.2 Tracking the abstract syntax model. In order to trace the amount of tokens assigned to a given *Place* instance, we may use the graph triggers described in Listings 2 (trigger *tokenConnected*) and 3. They update the global *numberOfTokens* array whenever *Token* instances are created and deleted; by combining that technique with the one we used in Fig. 12 we define the precondition pattern in Fig. 13 which includes a check condition on the equality of the value stored in the trace model and the one cached in the global array. By defining a *fall* trigger with this pattern, the system may detect when the user has changed the number of *Token* instances assigned to a *Place* instance *whose concrete syntax mapping is maintained.*

4.3.3 Creating the trigger. Finally, in Listing 6, we combine the two precondition patterns into a disjunction to provide a complex precondition pattern for the *synchTokens* trigger. The pattern call to *mappedPlaceFigure* is only used to ensure that the entire precondition pattern configuration corresponds to exactly one abstract-concrete-trace tuple. In the action sequence, we use

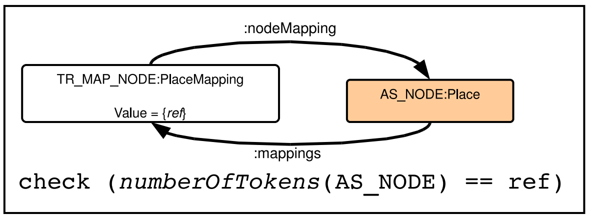


Fig. 13 Tracing attribute value changes in the abstract syntax

when-clauses to distinguish between the two operation modes. When synchronizing the attribute value change, the transformation computes the difference (`Diff`) in the number of tokens to the previously known value stored in the trace model (`value(TR_MAP_NODE)`) and proceeds to call a sub-routine which creates or deletes the necessary amount of tokens (`addOrRemoveTokens`). After that, the trace model is updated.

In the other case, when the abstract syntax model was changed, the value of the *numberOfTokens* array is simply copied into the concrete syntax attribute (`setValue(TR_MAP_NODE, numberOfTokens(AS_NODE))`). Finally, the trace model is updated with the new information.

```
/* Describes a connected
 * abstract syntax model
 *     - trace -
 *         concrete syntax model tuple */
pattern mappedPlaceFigure(AS_NODE,
                          TR_MAP_NODE,CS_NODE)=
{
 // abstract syntax
 Place(AS_NODE);
 // concrete syntax
 PlaceFigure(CS_NODE);
 // trace model
 PlaceMapping(TR_MAP_NODE);
 nodeMapping(_,TR_MAP_NODE,AS_NODE);
 mappings(_,AS_NODE,TR_MAP_NODE);
 diagramElements(_,TR_MAP_NODE,CS_NODE);
}

asmfunction numberOfTokens / 1;

@Trigger(sensitivity='fall', priority=2)
gtrule synchTokens() =
{
 precondition pattern pre() = {
  find mappedPlaceFigure(CS_NODE,AS_NODE,TR_MAP_NODE);
  find attributeTrace(CS_NODE,CS_ATTR,TR_MAP_NODE)
 }
 OR {
  find mappedPlaceFigure(CS_NODE,AS_NODE,TR_MAP_NODE);
  find tokenCountTrace(AS_NODE,TR_MAP_NODE);
 }
 action
 {
  when(update(value(CS_ATTR))) do seq {
   // attribute value in the
   // concrete syntax has changed
   let Diff = value(CS_ATTR) - value(TR_MAP_NODE) in
    call addOrRemoveTokens(AS_NODE,Diff);
   // update trace model
   setValue(TR_MAP_NODE, value(CS_ATTR));
  }
  when(update(numberOfTokens(AS_NODE))) do seq {
   // number of token instances in the
   // abstract syntax has changed
   setValue(CS_ATTR, numberOfTokens(AS_NODE));
   // update trace model
   setValue(TR_MAP_NODE, numberOfTokens(AS_NODE));
  }
 }
}

rule addOrRemoveTokens(Place,Diff) = seq
{
 let I = Diff in seq
 {
   if (i>0) try choose Tok
         with find placeToken(Place,Tok)
   do seq {
     // delete Tokens
     if (I == 0) fail;
     delete(Tok);
     update I = I - 1;
   }
   else seq {
    // create Tokens
    if (I == 0) fail;
    new (Token(Tok) in Place);
    new (tokens(_, Place,Tok));
    update I = I + 1;
}}}
```
Listing 6 Attribute value abstraction

4.3.4 A sample execution sequence. In Fig. 14, a sample execution sequence of the **synchTokens** trigger is shown (note that edge types have been omitted for the sake of simplicity). In Phase 1, the model is in a consistent state, where place *P0* contains a token *Tok0*, and this fact is reflected in the diagram model as a token count attribute value of 1, stored in *TC*.

Next, we follow the scenario where the user adds a token to P0 (Phase 2). As a reaction, the pattern matching RETE network assigned to the *connectedToken* graph pattern signals a new match, and the *tokenConnected* trigger is fired (Listing 2). As the slot assigned to P0 in the *numberOfTokens* global array is updated by trigger, the RETE network again signals a match set loss in the *tokenCountTrace* graph pattern (Fig. 13), which in turn fires the *synchTokens* trigger (Listing 6). As a result, first, the attribute value in the concrete syntax is updated (Phase 3), and finally the reference value in the trace model is modified (Phase 4).

4.3.5 Relevance In Sec. 4.3, we have demonstrated the flexible extensibility of our core approach. By combining basic mapping techniques from the *Mapping Library*, the language engineer is able to specify an arbitrary mapping between abstract and concrete syntax models. This way, the domain-specific visualization and editing framework can be directly adapted to any abstract syntax metamodel, without the need of constructing an intermediate language (Fig. 3).

Our approach can be extended to scenarios where the same abstract syntax model is mapped to multiple diagrams, which is a frequent requirement in advanced modeling environments. In that case, for n diagram types, n (bi-directional) mapping transformations have to be developed; whenever a change is made in one of the diagrams, the change is propagated to the abstract syntax and then automatically propagated further to those diagrams where the affected abstract syntax elements are displayed. This way, the consistency across multiple diagrams can be automatically preserved.

4.4 Implementation details

The authors have developed a complete implementation of the approach described in Sec.4, which is adapted to the ViatraDSM domain-specific language engineering framework. The *Mapping Library* (Fig. 3) consists of generic live transformation programs written in the VIATRA2 Textual Command Language, and are capable of facilitating a two-way, one-to-one correspondence mapping based on the trace metamodels described in Sec. 3.1.3.

Along with this implementation, we have also provided a prototype VIATRA2 import facility, which is able to process GMF specification models (.ecore, .gmfgraph, .gmfmap). The importer generates a domain-specific graphical editor for the ViatraDSM framework, which is functionally equivalent to the GMF editor.

The converted editor (the working example Petri net editor is shown in Fig. 15) works similarly to the original GMF editor; the user can place the same elements on the diagrams and edit the same attributes. However, ViatraDSM allows direct access to the full abstract syntax model (as shown in the Outline view on the right in Fig. 15), so it can be manipulated independently of the concrete syntax. The Mapping Library provides bi-directional synchronization between the two representations; it can be easily extended at runtime so that a custom mapping can be developed rapidly.

5 Related work

In this related work section, we provide a brief evaluation of leading commercial and academic initiatives in the field of domain-specific modeling frameworks, with a special focus on support for abstract–concrete syntax synchronization and model transformation support.

5.1 Model synchronization and traceability models

Our application of traceability is different from the traditional traceability applications to requirements management and tracking in model driven scenarios, described in many papers (e.g. [12,13]). In the current paper, traceability models are used to link two representations of the same modeling language (namely, abstract and concrete syntax) together, to drive a bi-directional synchronization transformation (conceptually similar to e.g. Fondement's work [14]). As emphasised, our contribution is two-fold: (i) we use event-driven live transformations to facilitate the automatic generation of trace models and

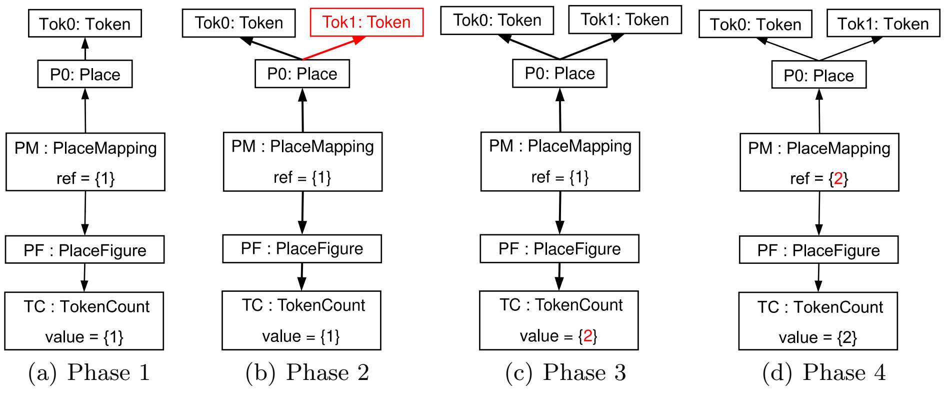
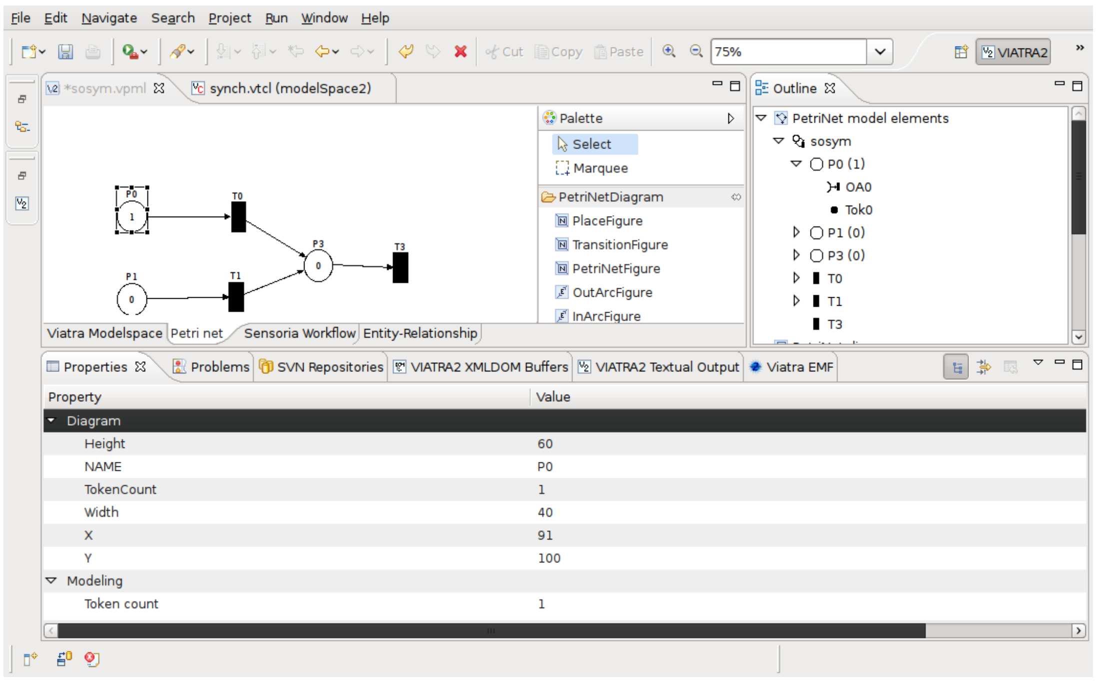


Fig. 14 Petri net synchronization execution phases



Property	Value
Diagram	
Height	60
NAME	P0
TokenCount	1
Width	40
X	91
Y	100
Modeling	
Token count	1

Fig. 15 The Petri net editor in ViatraDSM

the execution of the mapping, (ii) we use a generic approach to trace metamodels which enables the designer to choose how much information is contained in the trace models and how much mapping logic is (implicitly) implemented in the transformations themselves.

Event-driven techniques, which are the technological basis of live model transformations, have already been used in many other fields of computer engineering. In relational database management systems (RDBMS), even the concept of triggers [15] can be considered as simple operations whose execution is initiated by events. Later, event-condition-action (ECA) rules [16,17] were introduced for active database systems as a generalization of triggers, and the same idea was adopted in rule engines [18] as well. Specification of live model transformations is structurally and conceptually similar to ECA rules as discussed in Sec. 3.2. However, ECA-based approaches lack the support for triggering by complex graph patterns, which is an essential scenario in model-driven development.

In case of live transformations, *changes* of the source model are categorized as (i) an *atomic* model update consisting of an operation (e.g. create, delete, update) and operands (model elements); or, more generally, (ii) a complex *sequence* (set, transaction) of such atomic operations. To execute an incremental update, an atomic or complex model change has to be captured and processed. For this purpose, the following approaches have been proposed in case of *declarative transformation languages*:

– The Progres [19] graph transformation tool supports incremental attribute updates to invalidate partial matchings in case of node deletion immediately. On the other hand, new partial matchings are only lazily computed.

- The incremental model synchronization approach presented in [20] relies on various heuristics of the correspondence structure interconnecting the source and target models using triple graph grammars[21]. Dependencies between correspondence nodes are stored explicitly, which drives the incremental engine to undo an applied transformation rule in case of inconsistencies. Other triple graph grammar based approaches for model synchronization (e.g. [22]) do not address incrementality.
- In relational databases, materialized views, which explicitly store their content on the disk, can be updated by incremental techniques like Counting and DRed algorithms [23]. As reported in [24], these incremental techniques are also applicable for views that have been defined for graph pattern matching by the database queries of [25]. The use of non-materialized views have been discussed in [26].
- Triple graph grammar techniques are also used in [27] for tool integration based on UML models. The aim of the approach is to provide support for change synchronization between various languages in several development phases. Based on an integration algorithm, the system merges changed models on user request. Although it is not a live transformation approach, it could benefit from being implemented as such.
- QVT Relations [28] is the OMG standard language for capturing model transformations with specific focus on bidirectional transformations for incremental model synchronization. Concerning its expressiveness, the QVT Relations language uses a similar formalism to triple graph grammars (and our approach). However, none of the existing QVT tools support event-driven live execution.
- [11] proposes a more general solution where *fact addition* and *fact removal* constitute an elementary change. Since the underlying TefKat [29] tool uses a transformation engine based on SLD resolution, a fact change may represent atomic updates (involving a single operation) as well as more complex changes, since a fact may encode information about multiple model elements (such as a complex pattern describing a UML class with attributes). This approach is only applicable to fully declarative transformation languages, since incremental updates involve the processing and modification of the SLD resolution tree (which, in broad terms, can be thought of as a special structure storing the whole transformation context).
- [30] describes a special application of incremental updates for the consistency checking of UML models. The approach provides a rule-based formalism to specify well-formedness constraints which are evaluated instantly after model modifications. Our demonstrating example illustrates how specialised transformations can be applied to a similar problem, but on a higher abstraction level.
- Recently, [31] emphasized the use of *weaving models* as a special kind of correspondence models to semi-automatically derive model transformation rules for model synchronization. The authors present a metamodel-based method that exploits metamodel data to automatically produce weaving models in the AMW System. The weaving models are then derived into model integration transformations.
- Fondement's work shares a lot of concepts presented in our paper: a complete mapping metamodel with semantics is provided in [14] to support arbitrary mappings between abstract and concrete syntaxes of textual DSLs (transformation execution is carried out with the Kermeta tool). By his approach, mapping semantics is precisely defined for the model elements (e.g. sequence, alternation, iteration, template substitution rules); in our approach, the designer is free to choose how much semantic information is included in the mapping model and how much is implicitly defined in the transformation rules. Also, it is important to note that while the mapping transformations are incremental, they are not live, but executed in a recursive descent-type batch execution scheme.

5.2 Domain-specific modeling

A number of third-party tools have been developed to simplify the complicated GMF workflow. Exeed [32] and EuGENia are modern EMF-based domain-specific editor tools of the Epsilon [33] project. They aid the tool builder in the prototyping phase by significantly simplifying the task of creating an EMF-based reflective tree view editor (in the case of Exeed) or a simple GMF-based graphical editor (EuGENia), by using an annotated textual representation of ECore metamodels.

MetaEdit+ [34] is one of the first commercial DSM modeling frameworks with support for generating and customizing domain-specific editors and code generators. Since 1995, it has been applied successfully in various application domains. While MetaEdit+ supports multiple concrete syntax representations for a conceptual domain model, there is no automated support for working with multiple domains simultaneously within an integrated editor. Model transformations can be implemented by hand-coding using an API. On the MetaEdit+ website, there is an example, where the generated code is "back-annotated", to enable visual tracking while the debugger is stepping through the code.

Microsoft has also released its DSL tools [35] in order to support software factories [36]. New domain-specific languages are integrated into Visual Studio as plugins. While Microsoft DSL tools offer an advanced way for customizing the graphical representation, the framework is not based on a model transformation backend which leaves the language engineer with application programming as the only option to create model conversions, code generators or model mappings.

The Pounamu [37] is a meta-tool for multi-view visual language environment construction for Eclipse-based editors. The tool permits rapid specification of visual notational elements, underlying tool information model requirements, visual editors, the relationship between notational and model elements, and behavioural components. While offering a rich support for the generated editors, model transformations are currently not supported by the framework.

Xactium XMF-Mosaic is an integrated, Eclipse-based extensible development environment for domain-specific visual and textual languages [38]. It has support for domain metamodel specification with OCL constraints, and editor generation (concrete textual and graphical syntaxes can be supported for the same language). Using snapshots, prototyping a new language is accelerated considerably. However, unfortunately, as the generator feature is still in its infancy (e.g. constraints are not propagated to the generated source code), using advanced capabilities of the XMF platform require a considerable amount of manual coding effort.

There are also several DSM frameworks which are complemented with support for model transformations, typically, using a *graph transformation* [8] based approach, which approaches show the closest correspondence with our approach.

The TIGER project [39] (which is a conceptual continuation of the GenGEd [40] and DiaGen [41] tools from the 90s) primarily aims at generating syntax-directed editors as Eclipse plugins based upon the EMF and GEF technologies and the AGG graph transformation engine. Recent development focused on the tight integration of TIGER's model transformation infrastructure with Eclipse GMF; currently, TIGER is able to generate GMF-based editors with rich and complex editing facilities (such as the execution of a complex editing action based on a *single* graph transformation rule). However, currently TIGER only supports the execution of a single graph transformation rule, and thus, lacking a control flow language, complex transformations required for model synchronization can only be implemented using extensive Java-coding.

DiaMeta (a follow-up of DiaGen [41]) replaces hypergraph grammars by MOF as provided by the MOFLON tool suite [42] to allow users not only to specify domain-specific modeling languages but also to generate corresponding diagram editors. As DiaMeta focuses primarily on freehand editing, their contribution is complementary to ours as our main focus is syntax-driven editing.

The Generic Modeling Environment (GME) [43] combined with the GReAT model transformation engine [44] provides similar functionality to GMF outside the Eclipse world, with a static one-to-one mapping between abstract and concrete syntax models.

The VMTS [45] framework also provides support for domain-specific modeling and model transformations by providing plugins for Microsoft Visual Studio. It offers core editing functionalities comparable to Eclipse GEF, but further manual coding is required for visual syntax and editing functionality. Its distinguishing feature is an optimizing OCL processor for efficient handling well-formedness constraints. Since VMTS includes a powerful model transformation engine, it would be possible to implement an approach that is similar to ours. However, no such research has been published yet.

Most advanced multi-domain modeling features are supported by ATOM3 [46], which defines the concept of view metamodels sharing a common metamodel of a visual language. In [47], user-guided manipulation events are directly represented as model elements in the model store, while triple graph grammars [21] are extended to *event driven grammars* to determine the kind of event and the model elements affected. Change detection is directly linked to user interface events as this approach primarily targets (domain-specific) modeling environments. Note that this approach, *does not* rely on live transformations since the transformation context is not preserved; instead, the underlying ATOM3 [46] engine is started whenever an event from the UI is received.

6 Conclusion

As the core contribution of this paper, we presented a new approach for constructing syntax-driven domain-specific graphical editors. Building on this infrastructure, we provide high level support for the specification and efficient execution of live transformations which seamlessly maintain correspondence between completely separated abstract and concrete syntax representations. Our approach provides a scalable solution in terms of complexity, since language engineers can build on a generic Mapping Library to create custom mapping rules which focus strictly on those cases where customization is really necessary.

In the paper, we deliberately focus on solving model synchronization problems in DSM environments. It is important to note that our approach has been implemented and tested in more complex case studies than the Petri net example of the paper. We successfully applied the proposed mapping models in other application scenarios such as incremental well-formedness constraint evaluation (discussed in detail in [3]) where mapping models are used to indicate model contexts where a particular constraint is violated. The approach has also been applied to interactive model execution and design-time discrete simulation of DSMLs, elaborated as case studies in [5].

As a research contribution to the SENSORIA EU FP6 project, we have implemented a workflow model editor based on a customized version of the JBoss Process Definition Language for the SENSORIA Development Environment [48]. Additionally, the DSM editor was augmented with an incremental code generator applied in an exogenous transformation scenario, where the domain-specific process model contained in the editor was synchronized incrementally on-the-fly with the process description deployed in the JBoss jBPM workflow server. The mapping models used were based on the concepts described in the current paper – the case study has been reported in [49].

At the core of our approach is a novel event-driven, incremental execution scheme, which is based on the high performance incremental pattern matcher of Viatra2, capable of efficiently scaling up to 100000 model sizes on a desktop computer (for a detailed performance investigation and benchmarks see [50]).

– With incremental live transformations, complex language-specific constraints can be formulated as graph patterns [3]. Similarly to our synchronization transformations, model manipulation sequences are used to (i) signal constraint violations to the user on-the-fly as the errors have been introduced; and, (ii) corrective measures can be applied in a best-effort approach to correct models accordingly.
– Also, live transformations can be easily applied for model execution scenarios. In an integrated environment like ViatraDSM, such an infrastructure can support interactive model simulations where a user is allowed to edit models as they are being executed [5]. As the incremental pattern matcher provides instantaneous feedback, the user's editing actions affect the simulation state directly. This way, model execution can be efficiently supported at design-time, integrated into the syntax-driven editing environment.

As a main direction of future research, we intend to investigate how complex mapping semantics can be embedded into mapping metamodels. While our main target for the current paper was to match the mapping capabilities of GMF, a straightforward extension to the mapping metamodel would incorporate advanced mapping rules being specifiable for abstract-concrete syntax pairs (e.g. value abstraction as presented in Sec. 4.3). Such an approach would merge trace modeling contributions such as [31] with our domain-specific research context.

References

1. The Eclipse Project: Graphical Modeling Framework `http://www.eclipse.org/gmf`.
2. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems. Volume 4199 of Lecture Notes in Computer Science., Genova, Italy, Springer (October 2006) 321–335
3. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: Theory and Practice of Model Transformations. Volume 5063/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 107–121
4. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming **68**(3) (October 2007) 214–234
5. Ráth, I., Vágó, D., Varró, D.: Design-time Simulation of Domain-specific Models By Incremental Pattern Matching. In: 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). (2008)
6. Metacase: MetaEdit+ `http://www.metacase.com/mep/`.
7. The Eclipse Project: Eclipse Modeling Framework `http://www.eclipse.org/emf`.
8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
9. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag (2003)
10. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT'08, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)
11. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Proc. of 9th International Conference on Model Driven Engineering Languages and Systems (MODELS 2006). Volume 4199 of LNCS., Heidelberg, Germany, Springer Berlin (2006) 321–335
12. Champeau, J., Rochefort, E.: Model Engineering and Traceability. In: SIVOES-MDA Workshop. UML 2003 Conference (2003)
13. Walderhaug, S., Johansen, U., Stav, E., Aagedal, J.: Towards a Generic Solution for Traceability in MDD. In: 5th ECMDA Workshop on Traceability. ECMDA Conference (2006)
14. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Grard, S., Jzquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. Model Driven Engineering Languages and Systems, Springer LNCS **4199/2006** (November 2006) 98–110 DOI 10.1007/11880240_8.
15. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2001)
16. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: A rulebase of ADBMS features. In Sellis, T., ed.: Proc. of the 2nd International Workshop on Rules in Database Systems. Volume 985 of Lecture Notes in Computer Science., Glyfada, Athens, Greece, Springer (September 1995) 1–17
17. Alferes, J.J., Banti, F., Brogi, A.: An Event-Condition-Action Logic Programming Language. In: In the proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA 06), Springer (2006) 29–42

18. Seiriö, M., Berndtsson, M.: Design and implementation of an ECA rule markup language. In Adi, A., Stoutenburg, S., Tabet, S., eds.: Proc. of the 1st International Conference on Rules and Rule Markup Languages for the Semantic Web. Volume 3791 of Lecture Notes in Computer Science., Galway, Ireland, Springer (October 2005) 98–112
19. Schürr, A.: Introduction to PROGRES, an attributed graph grammar based specification language. In Nagl, M., ed.: Graph–Theoretic Concepts in Computer Science. Volume 411 of LNCS., Berlin, Springer (1990) 151–165
20. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of 9th International Conference on Model Driven Engineering Languages and Systems, (MoDELS 2006). Volume 4199 of LNCS., Springer (2006) 543–557
21. Schürr, A.: Specification of graph translators with triple graph grammars. Technical report, RWTH Aachen, Fachgruppe Informatik, Germany (1994)
22. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, New York, NY, USA, ACM (2007) 285–294
23. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: ACM SIGMOD Proceedings, Washington, D.C., USA (1993) 157–166
24. Varró, G., Varró, D.: Graph transformation with incremental updates. In Heckel, R., ed.: Proc. of the 4th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2004). Volume 109 of ENTCS., Barcelona, Spain, Elsevier (December 2004) 71–83
25. Varró, G., Friedl, K., Varró, D.: Graph transformation in relational databases. Journal of Software and Systems Modelling (2005) In press.
26. J. Jakob, A.K., Schürr, A.: Non-materialized model view specification with triple graph grammars. In A. Corradini, ed.: International Conference on Graph Transformations. Volume 4178 of Lecture Notes in Computer Science (LNCS)., Heidelberg, Springer Verlag (2006) 321–335
27. Simon M. Becker, Thomas Haase, B.W.: Model-based a-posteriori integration of engineering tools for incremental development processes. Software and Systems Modeling **4**(2) (May 2005) 123–140
28. The Object Management Group: Documents Associated With Meta Object Facility (MOF) 2.0 and Query/View/Transformation, V1.0 (2008) `http://www.omg.org/spec/QVT/1.0`.
29. The University of Queensland: The TefKat tool homepage `http://tefkat.sourceforge.net/`.
30. Egyed, A.: Instant consistency checking for the uml. In: Proceedings of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 381–390
31. Marcos Didonet Del Fabro and Patrick Valduriez: Towards the efficient development of model transformations using model weaving and matching transformations. Software and Systems Modeling (July 2008) Special section paper. Springer Berlin / Heidelberg. DOI 10.1007/s10270-008-0094-z.
32. Kolovos, D.S.: Editing EMF models with Exeed. Technical report, Department of Computer Science, University of York (2007) `www.eclipse.org/gmt/epsilon/doc/Exeed.pdf`.
33. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. Model Driven Architecture Foundations and Applications, Springer LNCS **5095/2008** (June 2008) 1–16 DOI 10.1007/978-3-540-69100-6_1.
34. Metacase: MetaEdit+ `http://www.metacase.com/mep/`.
35. Microsoft: DSL Tools `http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx`.
36. Greenfield, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools `http://msdn.microsoft.com/library/en-us/dnbda/html/softfact3.asp`.
37. Zhu, N., Grundy, J.C., Hosking, J.G.: Pounamu: a meta-tool for multi-view visual language environment construction. In: Proceedings of the 2004 International Conference on Visual Languages and Human-Centric Computing, Rome, Italy (September 2004) 254–256
38. Daniel Amyot, H.F., Roy, J.F.: Evaluation of Development Tools for Domain-Specific Modeling Languages. System Analysis and Modeling: Language Profiles, Springer LNCS **4320/2006** (December 2006) 183–197 DOI 10.1007/11951148_12.
39. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as Eclipse plug-ins. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA, ACM (2005) 134–143
40. Bardohl, R., Ehrig, H.: Conceptual model of the graphical editor GENGED for the visual definition of visual languages. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Proc. Theory and Application to Graph Transformations (TAGT'98). Volume 1764 of LNCS., Springer (2000) 252–266
41. Köth, O., Minas, M.: Generating diagram editors providing free-hand editing as well as syntax-directed editing. In Ehrig, H., Taentzer, G., eds.: GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, Berlin, Germany (March 25–27 2000) 32–39
42. Minas, M.: Generating visual editors based on fujaba/-moflon and diameta. Technical report, University Paderborn (2006) Proc. 4th Fujaba Days, pp. 35-42, Technical Report tr-ri-06-275.
43. GME: The Generic Modeling Environment `http://www.isis.vanderbilt.edu/Projects/gme`.
44. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the use of graph transformation in the formal specification of model interpreters. Journal of Universal Computer Science (2003)
45. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in vmts. In: Proc. GraBaTs 2004: International Workshop on Graph Based Tools, Elsevier (2004)
46. de Lara, J., Vangheluwe, H.: AToM3: A Tool for Multi-formalism and Meta-modelling. In Kutsche, R.D.,

Weber, H., eds.: 5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings. Volume 2306 of LNCS., Springer (2002) 174–188

47. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. Software and Systems Modeling **6**(3) (2007) 317–347
48. The SENSORIA EU FP6 Research Project: The SENSORIA Development Environment Homepage (2009) `http://svn.pst.ifi.lmu.de/trac/sct`.
49. Polgár, B., Ráth, I., Szatmári, Z., Majzik, I.: Model-based Integration, Execution and Certification of Development Tool-chains. In: 2nd ECMDA Workshop on Model-Driven Tool and Process Integration. (2009)
50. Bergmann, G., Ákos Horváth, Ráth, I., Varró, D.: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: ICGT2008, The 4th International Conference on Graph Transformation. (2008)