# Live Model Transformations Driven by Incremental Pattern Matching

István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
bergmann.gabor@gmail.com, okrosa@gmail.com
rath@mit.bme.hu, varro@mit.bme.hu

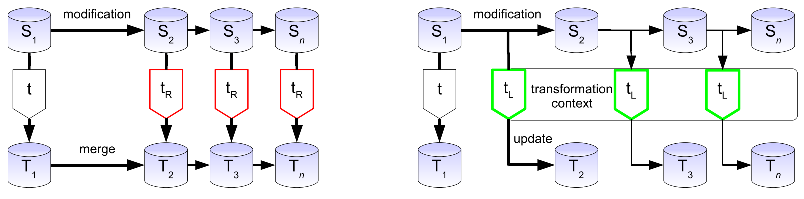
**Abstract.** In the current paper, we introduce a live model transformation framework, which continuously maintains a transformation context such that model changes to source inputs can be readily identified, and their effects can be incrementally propagated. Our framework builds upon an incremental pattern matcher engine, which keeps track of matches of complex contextual constraints captured in the form of graph patterns. As a result, complex model changes can be treated as elementary change events. Reactions to the changes of match sets are specified by graph transformation rules with a novel transactional execution semantics incorporating both pseudo-parallel and serializable behaviour.

## 1 Introduction

Model transformations play a crucial role in modern model-driven system engineering. Tool integration based on model transformations is one of the most challenging tasks with high practical relevance. In tool integration, a complex relationship needs to be established and maintained between models conforming to different domains and tools. This *model synchronization* problem can be formulated as to keep a model of a *source language* and a model of a *target language* consistently synchronized while developers constantly change the underlying source and target models. Model synchronization is frequently captured by *transformation rules*. When the transformation is executed, *trace* signatures are also generated to establish logical correspondence between source and target models.

Traditionally, model transformation tools support the *batch execution* of transformation rules, which means that input is always processed "as a whole", and output is always regenerated completely. However, in software engineering using multiple domain-specific languages, models are *evolving* and changing continuously. In case of large and complex models used in agile development, batch transformations may not be feasible.

*Incremental model transformations* address to update existing target models based on changes in the source models (called *target incrementality* in [1]), and to minimize the parts of the source model that needs to be reexamined by a transformation when the source model is changed (*source incrementality*). To achieve target incrementality, an incremental transformation approach creates "change sets" which are merged with



(a) Re-transformation (b) Live transformation

**Fig. 1.** Incremental transformation approaches

the existing target model instance. In order to efficiently calculate which source element may trigger changes (source incrementality), the *transformation context* has to be maintained which describes the execution state of the model transformation system (e.g. variable values, partial matches). Depending on whether this is possible or not, there are two main approaches to incremental transformations, as discussed in Fig. 1 (adapted from [2]):

– Systems employing *re-transformations* lack the capability to maintain the transformation context over multiple execution runs, thus the entire transformation has to be re-run on the modified source models. This approach generates either new output models which must be merged with existing ones, or change sets which can be merged *in-situ*. As noted in [2], since the transformation context is lost, a merging strategy has to be employed. This involves the computation of which model elements are involved in the change, and which elements should be left untouched by the transformation. Thus, the feasibility of this approach depends heavily on the trace information. For instance, in case of graph transformation [3], negative application conditions (NACs) may be used to forbid the execution of a transformation rule twice on the same source element. An intelligent re-transformation based model synchronization approach has been proposed recently for ATL in [4], which targets bidirectionality rather than incrementality.
– In contrast, *live transformations* maintain the transformation context continuously so that the changes to source models can be instantly mapped to changes in target models. Live transformations are persistent and go through phases of execution whenever a model change occurs. Similarly to re-transformations, the information contained in trace signatures is used in calculating the source elements that require re-transformation. However, as the execution state is available in the transformation context, this recomputation can be far more efficient.

**Related work in incremental transformations.** 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 [5] 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 [6] relies on various heuristics of the correspondence structure interconnecting the source and target models using triple graph grammars[7]. 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. [8]) 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 [9]. As reported in [10], these incremental techniques are also applicable for views that have been defined for graph pattern matching by the database queries of [11]. The use of non-materialized views have been discussed in [12].
- In [13], user-guided manipulation events are directly represented as model elements in the model store, while triple graph grammars [7] 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 [14] engine is started whenever an event from the UI is received. The idea, however, could be used in a live transformation environment.
- [2] proposes a more general solution where *fact addition* and *fact removal* constitute an elementary change. Since the underlying TefKat [15] 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).

**Contributions of the paper.** In the current paper, we present a novel approach to incremental model transformations based on incremental graph pattern matching and complex transaction handling. The main features of our contribution can be identified as follows: we support (i) atomic changes as well as model changes for complex constraints; (ii) various style of model transformation languages including fully declarative, partially declarative and procedural languages; and (iii) live transformations by preserving the transformation context. We discuss also how our incremental engine has been implemented and integrated as part of the VIATRA2 model transformation framework.

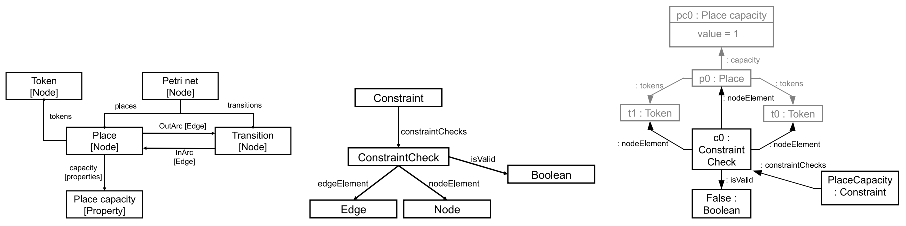
## 2 Preliminaries

In this section, we give a motivating example for live transformations. We also provide a brief introduction to the transformation language of the VIATRA2 framework.

## 2.1 Demonstrating example

In this section, we demonstrate the technicalities of our approach using Petri nets, which are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. From a system modelling point of view, a Petri net model is frequently used for correctness, dependability and performance analysis in early stages of design.

Fig. 2(a) shows a simplified metamodel for Petri nets (captured in the VPM formalism [16] of VIATRA2). Petri nets are bipartite graphs, with two disjoint sets of nodes: Places and Transitions. Places may contain an arbitrary number of Tokens. Tokens are also modeled as objects to support visual representation. The Petri net concept can be extended by the notions of *place capacity* constraints which impose a limit on the number of Tokens a Place can hold.



(a) Petri net metamodel (b) Marker metamodel for constraints (c) Model instances

**Fig. 2.** VIATRA metamodels and model instances

In the paper, we demonstrate our approach by the incremental validation of a complex dynamic modeling constraint for user editing events. In this use case, the user is editing models using a domain-specific editor which is capable of enforcing static type constraints so that only syntactically correct Petri net graphs can be produced. However, an advanced framework may go beyond this and provide immediate feedback if more complex constraints, such as a *capacity constraint* is violated (e.g. the user tries to assign too many tokens to a place).

In order to provide support for the editor, the modeling environment makes use of a *marker metamodel* which is a special type of trace model depicted in Fig. 2(b). A *Constraint* denotes a particular run-time constraint being enforced within the editor, e.g. “PlaceCapacity”. For each constraint, we explicitly mark all the (Petri net) elements, which are required to evaluate the constraint within a given context by a *ConstraintCheck* element. Each evaluation context of a *Constraint* is explicitly marked by a *ConstraintCheck* instance (i.e. separately for each Petri net place and its respective tokens in our case). The *isValid* relation indicates whether the constraint is valid *currently* for the context defined by the ConstraintCheck instance; the runtime environment makes use of this relationship to indicate graphical feedback to the user. In Fig. 2(c), place *p0* contains two tokens but has a capacity of 1, thus, the associated ConstraintCheck instance indicates that the PlaceCapacity constraint is violated in this context. In our demonstrating example used throughout the paper, we aim at providing an incremental evaluation of the capacity constraint in all contexts in response to elementary changes or complex transactions initiated by the user or another transformation.

### 2.2 Model transformations in VIATRA

The transformation language of VIATRA2 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) [3] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [17] 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 new (more complex) one. 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) [3] 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 Incremental pattern matching in VIATRA

Pattern matching plays a key role in the execution of VIATRA2 transformations. The goal is to find the occurences of a graph pattern, which contains structural as well as type constraints on model elements. In the case of incremental pattern matching, the occurrences of a pattern are readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences are stored, they can be retrieved in constant time – excluding the linear cost induced by the size of the result set itself –, making pattern matching a very efficient process. Besides memory consumption, the drawback is that these stored result sets have to be continuously maintained, imposing an overhead on update operations.

Our approach is based on the RETE algorithm [18], which is a well-known technique in the field of rule-based systems. This section is dedicated to giving a brief overview on how we adapted the concepts of RETE networks to implement the rich language features of the VIATRA2 graph transformation framework.

**Tuples and Nodes.** The main ideas behind the incremental pattern matcher are conceptually similar to relational algebra. Information is represented by a tuple consisting of model elements. Each node in the RETE net is associated with a (partial) pattern and stores the set of tuples that conform to the pattern. This set of tuples is in analogy with the relation concept of relational algebra.

The *input nodes* are a special class of nodes that serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing unary tuples representing the instances that conform to the type. Similarly, there is an input node for each relation type, containing ternary tuples with source and target in addition to the identifier of the edge instance. Miscellaneous input nodes represent containment, generic type information, and other relationship between model elements.

*Intermediate nodes* store partial matches of patterns, or in other terms, matches of partial patterns. Finally, *production nodes* represent the complete pattern itself. Production nodes also perform supplementary tasks such as filtering those elements of the tuples that do not correspond to symbolic parameters of the pattern (in analogy with the projection operation of relational algebra) in order to provide a more efficient storage of models.

**Joining.** The key component of a RETE is the join node, created as the child of two parent nodes, that each have an outgoing RETE edge leading to the join node.

The role of the join node can be best explained with the relational algebra analogy: it performs a natural join on the relations represented by its parent nodes.

Figure 3(a) shows a simple pattern matcher built for the *sourcePlace* pattern, which describes a Place-Transition pair connected by an out-arc, illustrating the use of join nodes. By joining three input nodes, this sample RETE net enforces two entity type constraints and an edge (connectivity) constraint, to find pairs of Place and Transitions instances which fulfill the constraints described in the pattern.

**Updates after model changes.** The primary goal of the RETE net is to provide incremental pattern matching. To achieve this, input nodes receive notifications about changes on the model, regardless whether the model was changed programmatically (i.e. by executing a transformation) or by user interface events.

Whenever a new entity or relation is created or deleted, the input node of the appropriate type will release an update token on each of its outgoing edges. To reflect type

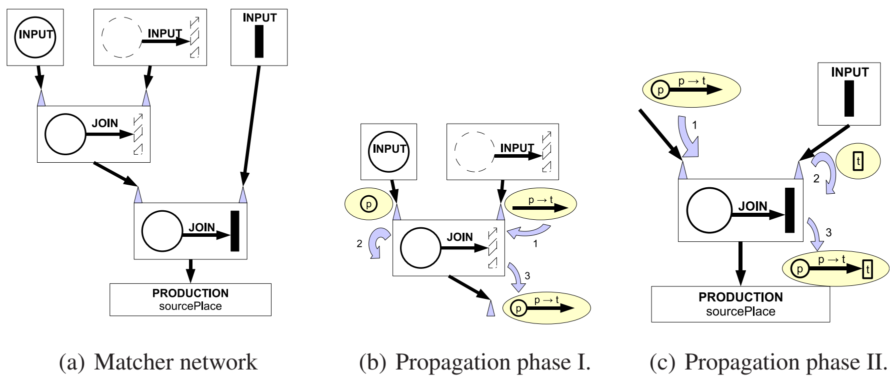


**Fig. 3.** RETE matcher for the sourcePlace pattern

hierarchy, input nodes also notify the input nodes corresponding to the supertype(s). Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set.

Each RETE node is prepared to receive updates on incoming edges, assess the new situation, determine whether and how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.

Figure 3(b) shows how the network in Fig. 3(a) reacts on a newly inserted out-arc. The input node for the relation type representing the arc releases an update token. The join node receives this token, and uses an effective index structure to check whether matching tuples (in this case: places) from the other parent node exist. If they do then a new token is propagated on the outgoing edge for each of them, representing a new instance of the partial pattern "place with outgoing arc". Fig. 3(c) shows the update reaching the second update node, which matches the new tuple against those contained by the other parent (in this case: transitions). If matches are found, they are propagated further to the production node.

More details of this incremental pattern matching approach can be found in [19]. It is worth pointing out that our RETE implementation significantly extends [20], the only existing RETE based approach in the field of graph (and model) transformation. In the future, we plan to incorporate another incremental approach [21] based on notification arrays to store a tree for partial matchings of a pattern.

## 4 Live transformations driven by incremental pattern matching

Based on our incremental pattern matching technology introduced in Sec. 3, we now propose a novel approach to live model transformations.

## 4.1 Overview of the approach

**Model changes.** 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 isomorphic to P.

Changes in the matching set can be tracked using the RETE network. 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 [2]. 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).

**Transformation context and efficient recomputation.** Live transformation execution requires the continuous maintenance of the *execution context* to avoid the necessity of model merging in target models. In our approach, this context contains:

– *global variables*, which are persisted to enable the transformation engine to store (global) cached values.
– *pattern variables*, which are maintained by the incremental pattern matching engine after each atomic model manipulation operation. 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.

As a result, the *computation* required to initialize and execute the incremental transformation sequence after a change is very efficient, since pattern matching, the most cost-intensive phase of the transformation, is executed instantly.

**Explicit specification.** In addition to targeting the incremental execution of model synchronization transformations, our approach is intended to support a broader range of live transformations. For this purpose, incremental transformation rules, called *triggers* are explicitly specified by the transformation designer. A trigger is defined in the form of a graph transformation rule: the precondition of its activation is defined in the form of a graph pattern, while the reaction is formulated by arbitrary (declarative or imperative) transformation steps.

In fact, not only tool integration, but many application scenarios can be formulated as incremental transformations, especially, in the context of domain-specific modeling such as (i) model execution (simulaton), where triggers may be used to execute the dynamics semantics of a domain-specific language; (ii) constraint management, where incremental transformations are used to check and enforce the validity of a complex constraint; (iii) event-driven code generation, where the textual representation of abstract models may be incrementally maintained as the source model changes.

### 4.2 Triggers

In our approach, the basic unit of incremental transformations is the *trigger*. 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(priority='10', mode='always', sensitivity='rise')
gtrule initPlace() = {
 precondition pattern pre(P) = {
  Place(P);
  Place.Place_Capacity(PC);
  Place.capacity(Cap,P,PC);
  neg pattern placeSet(P) = {
    Constraint.ConstraintCheck(CC);
    Constraint.ConstraintCheck.nodeElement(NE,CC,P);
  }
 }
 action {
   new(Constraint.ConstraintCheck(CC));
   new(Constraint.ConstraintCheck.nodeElement(NE, CC, P));
 }
}
```

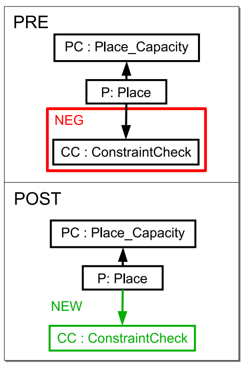


**Fig. 4.** Place instance initialisation

In Fig. 4, a simple trigger is shown. It is automatically fired after the user creates a new Place and the modeling environment creates (as a complex model change involving multiple elements) an additional Capacity and a ConstraintCheck marker element for the new Place-Place_Capacity pair. As a common technique in graph transformation based approaches, we use a negative application condition to indicate that the action sequence should only be fired for new pairs without a marker element.

This simple example highlights a number of extensions that constitute our additions to the VIATRA2 transformation language: the new `Trigger` annotation is used to indicate that the graph transformation rule should be executed as an event-driven transformation. The annotation uses the following options (specified in a Java-like syntax):

**Priority** (integer): Defines a precedence relation on multiple active triggers (triggers with higher priority value will run first).
**Mode** (always | once ): Defines whether a trigger is continuously scheduled for execution, or it is executed only once and then it becomes disabled.
**Sensitivity** (rise | fall | both): *Rise* triggers are activated whenever a new match is encountered; *fall* triggers are executed when a previously existing match is lost; *both* triggers execute on rises and falls as well.

### 4.3 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 *pat-*

*tern variables* (continuously maintained by the RETE network) and *persistent variables* (called *ASM functions* in VIATRA2; essentially global associative arrays).

```
// An array to cache token numbers
asmfunction numberOfTokens / 1;

@Trigger(priority='10', mode='always', sensitivity='rise')
gtrule placeAdded() = {
 precondition pattern pre(CC) = {
  Constraint.ConstraintCheck(CC);
  Place(P);
  Constraint.ConstraintCheck.nodeElement(NE_P,CC,P);
 }
 action {
   // Initialize the 'numberOfTokens' array
   update numberOfTokens(P) = 0;
   // calculate the initial number of tokens
  forall T with find placeToken(P,T)   do
    update numberOfTokens(P) = numberOfTokens(P)+1;
   // check the constraint's validity
   call constraintCheck(P,CC);
 }
}
```

**Listing 1.1.** Invoking constraint checking in the transformation context

In Listing 1.1, the *numberOfTokens* array is used in the persistent context to cache the amount of tokens assigned to a given place (the array is indexed by the Place reference). This trigger is fired after the ConstraintCheck marker element has been created by the trigger described in Listing 4, and performs the necessary steps to set up the cache with the appropriate value (Listing 1.2; note that some pattern definitions have been omitted for space considerations).

```
rule constraintCheck(  in P,  in CC) = seq {
  // match the PlaceCapacity element storing the value
  // of P's capacity.
  choose PC with find placeCapacity(P,PC)    do seq {
     if (numberOfTokens(P) <= value(PC))      seq {
       // delete a possible previous 'False' marking
       choose R find constraintFalse(CC,R)    do delete(R);
       // create a new 'True' marking
       new ConstraintCheck.isValid(R,CC, Boolean.True);
     }
     else seq {
       choose R with find constraintTrue(CC,R)    do delete(R);
       new ConstraintCheck.isValid(R,CC, Boolean.False);
     }
   }
}
```

**Listing 1.2.** Command sequence to check the validity of the capacity constraint

It is important to note that *pattern variables* (CC, P in the precondition, and T in patterns used in the action part) are also part of the maintained context, which makes the execution much more efficient. The underlying RETE-based pattern matcher maintains the matches for all involved patterns (*precondition*, *placeToken*, as well as *placeCapacity* and *constraintFalse* / *constraintTrue* in the constraintCheck rule) incrementally, thus the pattern matching operations (*forall* and *choose*, which pick all matches and one match, respectively) execute instantly, without any additional graph traversal.

## 4.4 Complex change detection

To detect complex model changes, the transformation developer can 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 similar to Listing 4 to connect the new Token to the CapacityConstraint marker element. In reaction to that, after initPlace has reached the commit point, the tokenAdded() trigger (Listing 1.3) is activated.

```
@Trigger(sensitivity='rise')
gtrule tokenAdded() = {
 precondition find connectedToken(P,CC,T) = {
  find placeToken(P,T);
  Constraint.ConstraintCheck(CC);
  Token(T);
  Constraint.ConstraintCheck.nodeElement(NE_Tok,CC,T);
 }
 action {
   update numberOfTokens(P) = numberOfTokens(P) + 1;
   call checkConstraint(P,CC);
 }
}
```

**Listing 1.3.** Trigger to handle the addition of Tokens

The *tokenAdded()* trigger updates the *numberOfTokens* array stored in the execution context, and initiates a constraint update which provides feedback to the user.

**Deletions.** To detect deletions, a trigger for the same precondition pattern as used in Listing 1.3 can be used in *fall* mode. In this case, the `undef` constant is assigned to the corresponding pattern variables to indicate that the model element identified by the pattern variable is no longer existent (Listing 1.4). However, other pattern variables (pointing to existing model elements) can be used in the action part in the usual way.

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

**Listing 1.4.** 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 *PlaceCapacity* property node is used to store the actual value of the capacity of the connected *Place*.

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

@Trigger(sensitivity='fall')
gtrule capacityChanged() = {
  precondition pattern pre(P,PC) = {
    find placeCapacity(P,PC);
    // check condition to define a value constraint
    check(value(PC) == capacities(PC))
  }
  action {
    // check whether the attribute update caused the activation
    if (PC!= undef && P!= undef && value(PC) != capacities(PC))     seq {
     // update constraint validity
     choose CC with find placeConstraint(P,CC)    do call checkConstraint(P,CC);
     // store new value
     update capacities(PC) = value(PC);
    }
  }
}
```

**Listing 1.5.** Handling attribute updates

In Listing 1.5, 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 *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 constraint validity.

## 4.5 Transaction management

In order to be able to perceive changes in the match set of a pattern over a complex model manipulation operation, such as the execution of a graph transformation rule or a complex editing operation, the model management system has to support transactions. A *transaction* is defined as a sequence of atomic model manipulation operations (e.g. create node, edge, instance-type-supertype relation, update attribute, etc.), followed by a commit command. The VIATRA2 framework ensures that all model manipulation occurs within a transaction.

The operational workflow of the live transformation system is shown in Fig. 5 from the viewpoint of transactions. After a transaction has reached its commit point, the system evaluates the changes in the match sets of precondition patterns of triggers registered in the trigger queue. Since the RETE networks are updated after each atomic model manipulation operation, a match set may experience transient changes while a long transaction is running. In our approach, only the *effective* changes are considered; thus, even if a new match is generated while a transaction is running, if that match is subsequently lost, the system will not process it for triggers. This mechanism is provided by the matching set *delta monitor*, which computes the net changes that occured during a transaction. After the changes have been evaluated, the execution engine processes triggers registered in the *trigger queue* and selects those with a precondition activated by the processed matching set changes, and prepares them for execution based on the current *execution mode*.

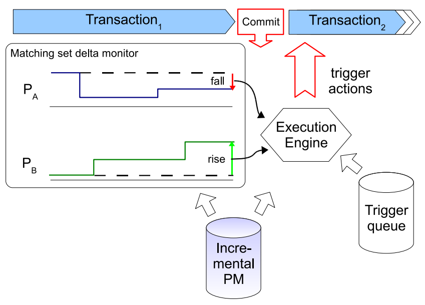


**Fig. 5.** Overview of incremental execution

**Execution modes.** Action sequences of activated triggers can be executed in two modes (Fig. 6). In the depicted scenario, we assume that there are three active triggers (T1–3) with their action sequences (AS1–3 respectively). After a transaction, the system encounters a new match (M(T1)–M(T3)) for each of the three triggers.

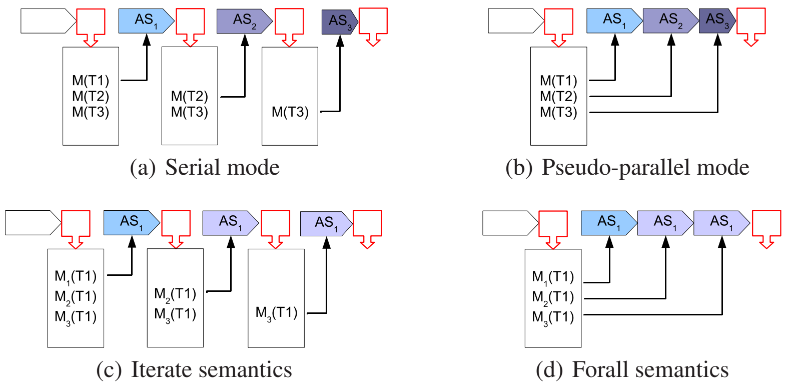


**Fig. 6.** Execution semantics for multiple match set changes and execution modes for multiple trigger activation

In *serial mode* (Fig. 6(a)), the action sequences are executed in separated transactions according to the priority order. After each commit point, the system re-evaluates all trigger conditions. In this mode, *conflicts* between competing triggers are eliminated (since the checks may reveal, for instance, that M2 was invalidated while AS1 was executed). However, a circular activation of triggers may result in infinite loops in case of serial execution mode.

In contrast, *pseudo-parallel mode* (Fig. 6(b)), action sequences are executed in a single transaction with a common commit point. In this case, conflicts may occur, and they need to be accounted for by the transformation designer. On the other hand, the execution is faster than in serial mode, since no intermediate checks are performed.

A similar race condition may arise for *multiple matches* for a single trigger. In Figures 6(d) and 6(c), trigger T1 has been activated for matchings $M_1$ - $M_3$. In *iterate mode*, we non-deterministically select one match, and execute its action sequence as a separate transaction. Then, if the rest of the matches are not invalidated, their respective actions are also executed one by one in separate transactions. In *forall mode*, all execution occurs in a single transaction with the possibility of conflicts which may cause a run-time error.

## 5 Conclusion

In the current paper, we presented a novel approach to live model transformations based on incremental graph pattern matching and complex transaction handling. Compared to existing incremental transformation approaches, the main added value of the current paper is (i) to preserve full transformation context in the form of pattern matches; (ii) to incorporate incremental reaction to complex model changes (both deletion and addition), and (iii) to provide incremental support for both declarative and imperative transformations with the help of complex transaction handling mechanism. Our approach is fully implemented and integrated to the VIATRA2 model transformation framework.

By using our live transformation engine, we carried out several case studies, mainly in the context of domain-specific modeling languages, including (i) incremental evaluation of complex constraints during user-guided model editing, (ii) discrete-event based model simulation captured by live transformation rules, and (iii) incremental code generation from domain-specific models. These case studies indicated the high potential of applying live transformations for different modeling problems.

## References

1. K. Czarnecki and S. Helsen: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–645
2. 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
3. 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)
4. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, New York, NY, USA, ACM (2007) 164–173
5. 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
6. 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

7. Schürr, A.: Specification of graph translators with triple graph grammars. Technical report, RWTH Aachen, Fachgruppe Informatik, Germany (1994)
8. 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
9. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: ACM SIGMOD Proceedings, Washington, D.C., USA (1993) 157–166
10. 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
11. Varró, G., Friedl, K., Varró, D.: Graph transformation in relational databases. Journal of Software and Systems Modelling **5**(3) (September 2006) 313–341
12. 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
13. 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
14. 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
15. The University of Queensland: The TefKat tool homepage `http://tefkat.sourceforge.net/`.
16. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (October 2003) 187–210
17. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag (2003)
18. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17–37
19. 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) Submitted.
20. Bunke, H., Glauser, T., Tran, T.H.: An efficient implementation of graph grammars based on the RETE matching algorithm. In Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: Graph-Grammars and Their Application to Computer Science. Volume 532 of Lecture Notes in Computer Science., Springer (1990) 174–189
21. Varró, G., Varró, D., Schürr, A.: Incremental graph pattern matching: Data structures and initial experiments. In Karsai, G., Taentzer, G., eds.: Graph and Model Transformation (GraMoT 2006). Volume 4 of Electronic Communications of the EASST., EASST (2006)