CONTEXTUAL GRAPH TRIGGERS

Gábor BERGMANN
Advisor: Dániel VARRÓ

I. Introduction

Model transformations play a crucial role in modern model-driven system engineering. Tool integration based on model transformations is a challenging task of high practical relevance, as it aims to harmonize the entire toolchain into a streamlined workflow. In tool integration scenarios, 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*.

Graph transformation (GT) [1] is a mathematical formalism frequently used for model transformation and other purposes; models are represented as (typed, attributed) graphs, and model manipulation is specified by *graph transformation rules*. GT rules consist of two *graph patterns*, and describe a change in the graph where an occurrence of the first pattern (precondition or left-hand-side, LHS) is replaced with the occurrence of the other pattern (postcondition or right-hand-side, RHS).

Traditionally, model transformation tools support the *batch execution* of transformation rules, which means that the input model is processed "as a whole", and output is either generated again from scratch, or, in more advanced approaches, updated using trace information from previous runs. However, models are *evolving* and changing continuously. In case of large and complex models used in agile development, batch transformations may not be feasible. To address this problem, live transformation [2] is an execution mode where changes to source models can be instantly mapped to changes in target models. Live transformations are persistent and go through event-driven phases of execution whenever a model change occurs.

Live transformation is based on event-driven rules. Some approaches including [2] regard only elementary model changes as events. Using a GT-based approach as a more general formalism of live transformations, [3] presented the novel formalism of *graph triggers* to capture high-level change events. Graph triggers are annotated GT rules that are executed upon the appearance or disappearance of pattern matches. These macroscopic events drive the execution of the transformation, independently of the low-granularity individual change that completed it. While monitoring the applicability of graph triggers is an involving task, it can be implemented efficiently with *incremental pattern matching* techniques. Apart from live synchronization, triggers can also be applied to detect and react upon complex details, e.g. on-the-fly well-formedness checking in domain-specific visual languages.

Experience with designing live transformations has shown that the solution introduced in [3] is too restrictive. It only allows the detection of a single pattern match (dis)appearance; neither simultaneous match set changes nor static context can be expressed to restrict the application of the trigger. This paper aims to introduce a significantly more general formalism for specifying graph triggers.

II. Background

The central concept of GT is the notion of graph patterns, which are basically small graphs. Pattern matching is the (computationally complex) process of identifying subgraphs in the model that correspond to the pattern. More formally, the pattern contains a set of *pattern variables* with some

constraints attached to them; a pattern match is a mapping of all pattern variables to model elements so that the image of the variables observe all constraints. The most important constraints are entity constraints stating that a variable be a node of a certain type, and relation constraints stating that a variable be an edge of a certain type, connecting two given variables. Some advanced formalisms, like the pattern language [4] of the tool VIATRA2 may permit disjunction and negation of constraints, as well as equality and inequality, attribute constraints, or *pattern composition*. The latter means patterns *call*ing each other; a pattern call constraint may prescribe that given pattern variables be mapped into a match of a given graph pattern. Pattern composition facilitates reusing common patterns, improves expressiveness and pattern matching performance in some cases, and may even be used recursively under certain circumstances.

As an illustration, a GT rule of a simplified Object-Relational Mapping (ORM), that maps object-oriented classes into database tables, is shown in Figure 1. The LHS pattern matches classes that do not have an associated table, and the RHS shows a table that is traced back to the same class. Executing the GT rule results in mapping an unmapped class to a table. Here the class node (C), the table (T) and the traceability edge (r) are pattern variables; their types (class, table, trace respectively) and configuration are constraints in both patterns, and the *NEG* box expresses negation. ORM can also be executed as a live transformation; if the GT rule is triggered whenever a match of the LHS appears, new classes will automatically be transformed into tables.

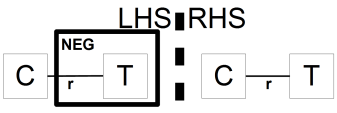

Figure 1: GT rule

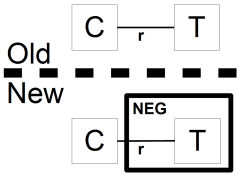

Figure 2: GGCP

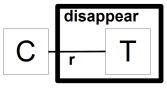

Figure 3: CP

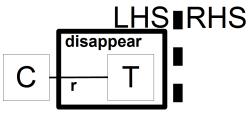

Figure 4: CGT

The aim to execute transformations without the costly re-evaluation of unchanged parts of the evolving source model is called source incrementality. Source incrementality can be achieved by employing incremental pattern matching techniques; for example, the RETE [5] incremental algorithm was used in [6]. The central idea of incremental pattern matching is that 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. There are two important drawbacks; one of them is the increased memory consumption due to the stored occurrence sets. Additionally, these stored result sets have to be continuously maintained whenever the model is changed, causing an overhead on model manipulation. Nevertheless, benchmarks [7] and practice have shown that incremental pattern matching can improve performance or scalability by up to several orders of magnitude in certain scenarios. Moreover, incremental pattern matching leads to easy discovery of appearing or disappearing pattern matches, thus it can be used to efficiently implement graph triggers.

III. General Graph Comparison Patterns

Triggers are essentially meant to make sense of the difference of two snapshots of the model (e.g. the state before and after a transaction), and to detect a high-level concept of events. The power of conventional graph triggers is restricted because their triggering condition (and LHS) does not have full access to the model in the two snapshots, but only to one element of the change set of pattern matches. For instance, if the ORM example needs to be a *bidirectional* live transformation, then new classes should be transformed into tables, while the deletion of the table should result in the deletion of the class; and likewise for mapping tables back to classes. The conventional trigger formalism cannot dif-

ferentiate between these cases without auxiliary structures; the previous example trigger would simply recreate a deleted table. To remedy this issue, I introduce the concept of *General Graph Comparison Pattern* (GGCP), which expresses the changes and unchanged parts between two versions of the graph, in graph pattern form.

Definition. Like conventional patterns, GGCPs also contain pattern variables that are to be mapped to graph elements. Each GGCP contains two sets of constraints, permitting all mentioned constraint types (including calling conventional patterns); one set should be satisfied in the old model, while the other is valid for the new model. As a special constraint type not belonging to either set, GGCPs can call each other. All four kinds of constraints can be the subject of disjunction or negation.

Semantics. A match is a constraint-satisfying mapping from GGCP variables to the union of graph elements present in the old and new model. The identity semantics are the following: elements existing both in the old and new model preserve their identity, i.e. a single variable will represent the same element in both snapshots for the purposes of the old and new constraint set. For elements that were created or deleted, and therefore only exist in one of the snapshots, basic constraints will only evaluate to true in one of the sets; their non-existence in the other snapshot can be explicitly checked using the negation of a sufficiently general constraint in the corresponding set. GGCP composition has the usual semantics.

Discussion. This formalism is powerful in the sense that it has unrestricted access to both the state before and the state after the change, with the full expressiveness of graph patterns. As a demonstration, Figure 2 shows a GGCP for bidirectional ORM, that detects deleted tables (in order to delete the class), but excludes the creation of a class. The downside is that compared to the less powerful graph trigger formalism, GGCPs capture changes on a low level of abstraction, therefore they are not as intuitive as more difficult to specify for a range of practical applications. An additional problem is that it is not apparent how an implementation can efficiently recognize GGCPs, without actually storing an archive copy of the old state of the graph, which would be costly both in terms of memory consumption and runtime performance.

IV. Change Patterns

To solve the limitations of conventional graph triggers and GGCPs, I propose the formalism of *Change Pattern*s (CP), inspired by Event-Condition-Action systems (e.g [8]), having access both to the high-level dynamic events happening between states and their static context in the new state, but without the costly direct access to the old snapshot. Thus CPs are high-level trigger guards, that are easy to specify, efficient to implement and powerful at the same time.

Definition. CPs also capture changes between two snapshots of a model. Each CP contains variables (with the same identity semantics as GGCPs) and one set of conventional constraints (including regular pattern calls) that are valid in the new state, to capture the static context of dynamic events. This set of constraints is extended by event queries, which consist of a sign (appearance / disappearance) and a pattern call, each of them similar to a conventional graph trigger guard. Finally, CPs are also allowed to call other CPs. All three kinds of constraints can be the subject of disjunction or negation.

Semantics. A match is a constraint-satisfying mapping from CP variables to the union of graph elements present in the old and new model. Conventional constraints have to be valid at the new state of the graph; they are always false for deleted elements, which can be explicitly detected using negation. Appearance events are satisfied iff the variables are mapped into a match of the called pattern in the new state that was not valid in the old snapshot. Conversely, disappearance events map the variables into matches in the old snapshot that are no longer valid in the new one. CP composition has the usual semantics.

Discussion. I argue that due to the higher granularity of capturing changes as appearance or disappearance of complex patterns, CPs are more intuitive and easier to specify than GGCPs for a vast

range of practical cases. Furthermore, they retain the efficiency of conventional graph triggers in most cases, as there is no need to keep an archive copy of the old model; it is sufficient to remember the delta of the match set of involved patterns. The latter task is easily accomplished by the incremental pattern matcher; as pattern matches can be rare and typically there is only a small number of appearing / disappearing matches as a result of one transaction, the associated costs are expected to be small.

It is worth noting that in addition to these these advantages in typical cases, CPs actually have the same expressive power as GGCPs. Basically GGCPs impose logical connectives (conjunction, disjunction, negation) on asserting (conventional) pattern constraints to be true only in the new state, only in the old state, in both states, or in neither one. As the constraint can be wrapped into a graph pattern and thus usable in event queries, CPs can also distinguish these four cases, respectively by imposing the appearance of the constraint, the disappearance, the presence in in the new state and negating the appearance, and negating both the presence in the new state and the disappearance. Since logical connectives are also available in CPs and GGCP calls can be translated into CP calls by induction, there is an equivalent CP for each GGCP. As a demonstration, Figure 3 shows the CP equivalent of Figure 2.

V. Contextual Triggers

Triggers defined with a CP as a guard (instead of a conventional pattern with an appearance / disappearance sign) are *Contextual Graph Triggers* (CGT). For instance, the CP on Figure 3 and an empty RHS form a CGT that propagates the deletion of tables to the associated classes (Figure 4). CGTs are more expressive than the graph triggers introduced in [3], as detecting events (match set changes) can be extended with the context of the change, consisting of a static pattern or even simultaneously occurring events. With exactly one event query and no static context, contextual triggers degenerate into conventional ones. With no event query (just a static pattern), contextual triggers degenerate into regular GT rules, that are applicable based on the current state alone, regardless of history.

Conclusion. The newly introduced CGT formalism unifies the advantages (expressiveness, ease of use, efficiency) of several approaches to achieve live transformation. Future work includes dealing with multiple simultaneous trigger activations (e.g. with priorities, conflict resolution), and finding a more direct approach to detecting attribute changes.

References

[1] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools, World Scientific, 1999.

[2] D. Hearnden, M. Lawley, and K. Raymond, "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)*, vol. 4199 of *LNCS*, pp. 321–335, Heidelberg, Germany, 2006. Springer Berlin.

[3] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró, "Live model transformations driven by incremental pattern matching," in *Proceedings of 1st International Conference on Model Transformation*, LNCS. Springer, 2008.

[4] A. Balogh and D. Varró, "Advanced model transformation language constructs in the VIATRA2 framework," in *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, 2006.

[5] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, 19(1):17–37, September 1982.

[6] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró, "Incremental pattern matching in the VIATRA model transformation system," in *Graph and Model Transformation (GraMoT)*, G. Karsai and G. Taentzer, Eds. ACM, 2008.

[7] G. Bergmann, A. Horváth, I. Ráth, and D. Varró, "A benchmark evaluation of incremental pattern matching in graph transformation," in *International Conference on Graph Transformation*, 2008.

[8] J. J. Alferes, F. Banti, and A. Brogi, "An event-condition-action logic programming language," in *JELIA*, M. F. et al., Ed., vol. 4160 of *Lecture Notes in Computer Science*, pp. 29–42. Springer, 2006.