Find-Replace as a Service of Graph-Based DSL Tool Development Framework

Elina Kalnina[1], Dmitrijs Kosarevskis[1] and Agris Sostaks[1]

[1]*Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia*

Abstract

The tool support of modelling languages including graphical domain-specific languages is not at the level tool users have got used to in other areas. Various features typical in other tools are missing in modelling tools, for example, find, find-replace and others. We discuss principles of concrete syntax-based find-replace for graph-based languages. We show how to add find-replace as a service of graph-based DSL tool development framework. The approach makes find-replace available to any language built with the framework.

Keywords

Domain-Specific Languages, Find-Replace, Domain-Specific Modelling Languages, DSL Tool development frameworks, Concrete Syntax-Based Model Transformation

1. Introduction

Probably one of the main reasons why MDE (Model Driven Engineering) has not reached the expected acceptance in the industry is the poor quality of tool support [1]. The features of limited support in MDE tools include find and find-replace typical in the modern user interface. In this paper, we consider one specific type of MDE tools – tools for graphical Domain-specific modelling. Domain-specific languages (DSL) is the area of MDE with the widest acceptance in the industry. We consider only graph-based or graphical DSLs. Authors believe that there are domains where graphical DSLs are more understandable and user-friendly than textual ones. However, the state of the art tool support for graphical DSLs is worse than for the textual ones. There are various DSL frameworks available facilitating the creation of a tool for a graphical DSL, for example, MetaEdit+ [2], Microsoft DSL Tools [3], Sirius (Sirius Desktop and Sirius Web) [4], ajoo [5], AToMPM [6] and many others. Typically, these frameworks contain means to define a DSL. For a graphical DSL, it is required to define abstract (domain metamodel) and concrete syntax (appearance of the language elements) and also the links between these syntaxes. The definition of a DSL is used to generate an editor for the DSL (e.g., Microsoft DSL Tools), or the definition is interpreted (e.g., Sirius, ajoo). A definition of a DSL is used also to provide specific services for the DSL editor, e.g., copying, cutting diagram elements, dialogues for the input of property values, tools to create diagram elements, etc. However, not all services typical in other tools (text editors, programming tools, etc.) are supported. For example, there is limited support for find and find-replace. Sirius Web [4] does not have the support of find and find-replace, only browser provided textual find is available. Sirius Desktop [4] supports

FPVM'21: 1st International Workshop on Foundations and Practice of Visual Modeling, June 21–25, 2021

textual find and find-replace working in the textual representation of a single file. Metadit+ [2] does not have find service. It has the replace menu, but it has different semantics. Actually, it is an instance copier to uncouple references. Microsoft DSL Tools [3] support textual find and find-replace in the XML representation of models. AToMPM [6] does not have the support of find and find-replace, but it integrates concrete syntax-based model transformation language. Find and find-replace working only in textual parts of a DSL is not enough for a language based on typed attributed graphs. The context in terms of element types and their attributes is also important.

The concrete syntax-based find as a service of DSL tool development framework has been proposed [7, 8]. We demonstrate how to extend concrete syntax-based find to support concrete syntax-based find-replace as a service of graphical (graph-based) DSL tool development framework. We add find-replace to the DSL tool development framework ajoo[5]. However, it is possible to apply the approach to other DSL tool development frameworks as well.

The motivation and find-replace principles are given in Section 2. The simple find-replace patterns and their semantics is discussed in Section 3. How the find-replace is added to the tool is discussed in Section 4. The potential use cases are discussed in Section 5. The related work is reviewed in Section 6.

2. Find-Replace Principles

A typical use case of the find-replace in office applications (text editors, spreadsheets) is to help maintain the consistency of a document. For example, to automatically correct all misspellings of a word or to replace a full name of an organization with an acronym. Replace the American version of a word with the British version of a word, e.g., modeling with modelling. Depending on the task, it is possible to replace all occurrences or only some occurrences. In fact, the find-replace feature in office applications is more powerful than most users are aware of. Besides simple textual find, tools support regular expressions, invisible symbols, formatting conditions, etc. However, for most users, it is enough with simple textual find-replace.

In the paper, we consider graphical or graph-based diagrams. The question is, what is find-replace for a graph-based diagram? Of course, we could use traditional textual find-replace, but this way it is not possible to restrict find query to look for a match only in certain property of a specific element type. Besides, the user may want to modify not only textual properties but also the structure of a diagram. In the general case, find-replace for graph-based languages is a special type of graph/model transformation - unidirectional in-place transformation. It should be possible to execute transformation for all or some manually selected matches only and to distinguish the find part from the replace part of the transformation.

The purpose of introducing find-replace is to improve the usability of tools, built using a DSL tool definition framework. We propose to provide find-replace as a universal service of a DSL tool development framework to enable find-replace in any graph-based DSL tool built using the framework. The potential users of find-replace are modellers working with a DSL. Users who can create models in a DSL should be able to use find-replace to modify models in a DSL, as well. The potential users are familiar with the concrete syntax of a DSL. Most probably they are not familiar with the abstract syntax of a DSL. Therefore, we propose to use concrete

syntax-based find-replace instead of traditional model transformation languages.

Of course, for complicated find-replace tasks, traditional model transformations can be used. However, there are a few things to consider:

- Typically, model transformation languages use the abstract syntax of the model. Therefore, to write a model transformation the user should be familiar with the encoding of models used by the DSL tool definition framework.
- The user should be familiar with the model transformation language.
- The transformation language should work in the technical space of the DSL tool development framework. Various technical spaces or repositories are used by DSL tools, e.g., Sirius [4] use EMF/Ecore [9], ajoo [5] use MongoDB [10], MS DSLs [3] use XML as data store and XSD as schema. The only technical space from these with appropriate model transformation support is EMF/Ecore [9].

There are a few concrete syntax-based model transformation approaches available. We discuss them in detail in Section 6. However, none of the approaches has been integrated with a DSL tool definition framework to support find-replace. We in contrast use the definition of a DSL in a DSL tool definition framework to provide a DSL specific find - replace transformation service. The concrete syntax of a DSL is used to define find-replace transformation.

The concrete syntax-based find-replace is based on the concrete syntax-based find [7, 8]. Any DSL diagram or a fragment of a DSL diagram can serve as a find query in the concrete syntax-based find. In the find-replace, however, there are two fragments of a DSL diagram: fragment used as find query and fragment describing a result of a replace operation, similarly to find-replace in office applications. However, to support more complicated transformations, the relation between find and replace patterns is required. We call it the find-replace edge. The semantics of the find-replace edge and patterns is described in Section 3.

Similarly to the concrete syntax-based find, any DSL diagram can be used as the find-replace diagram. However, to draw a find-replace pattern, find-replace elements (find-replace edge) should be enabled in a palette of a DSL diagram. After a find-replace pattern has been created it is possible to execute it. The find-replace is executed in two steps. The first step is to execute the find query. The result of the find query is a list of matches grouped by diagrams. The execution of the replace step is supervised. The user may select to perform replace operation for a specific match, all matches in a diagram or for all matches in all diagrams. Anyway, the replace operation for each match is executed separately. The activity diagram describing the usage of the find-replace is given in Fig. 1.

3. Find-Replace Semantics

When defining find-replace transformation, it is required to distinguish the fragment to be found and the replace fragment describing how the result is obtained. We call these fragments of a diagram - patterns - a find pattern (or left-hand side) and a replace pattern (or the right-hand side), both together find-replace pattern. Both sides of the find-replace pattern use the same language. The find-replace edge is used to define the correspondence between the left-hand

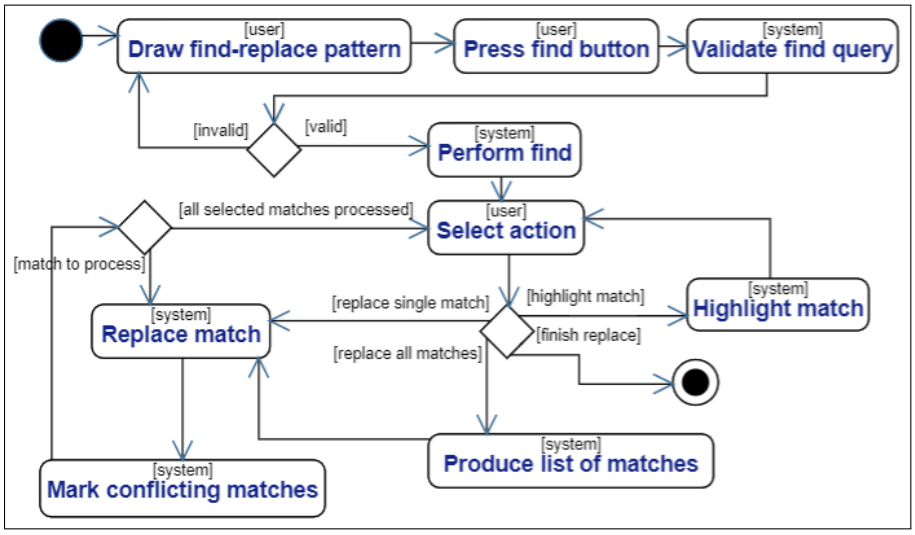

Figure 1: Find-replace usage scenario.

side and the right-hand side. To distinguish between multiple find-replace edges, it is possible to give a name to a find-replace edge. (The examples are given in Section 5.)

The find-replace is executed in two steps - find and replace. Before the execution of the find step, the find-replace diagram is validated. The diagram is considered invalid, if it is not possible to distinguish the left-hand side (the find pattern) and the right-hand side (the replace pattern) of the diagram (e.g., circular usage of the find-replace edge is detected), if the expressions are incorrect, etc.

A DSL element (node) at the source of a find-replace edge is used as a starting point for a find query. The whole pattern related to the node is matched. A set of elements to be matched may be restricted using constraints on attribute values. A constraint on string type attribute is matched using substring semantics. For other attribute types, equivalence is used. A match should satisfy all constraints defined by left-hand side patterns and attribute values. If there are multiple unrelated sub-graphs at the source of the find-replace diagram, each subgraph is matched separately. If there are multiple matches of the subgraphs in the diagram, the Cartesian product is used to produce a set of find results.

The result of the find step is a list of diagrams and a list of matches in a diagram (see Fig. 4). Afterwards, a match or multiple matches may be selected for execution of the replace operation. It should be noted that there may be overlapping matches. When the replace operation for a match is executed, all overlapping matches are marked as conflicting. It is not possible to execute a match marked as conflicting. If the user selects to replace all matches in a diagram, then replace operation is executed for each match. Matches marked as conflicting are skipped. It is also possible to replace all matches in all diagrams. In this case, the process of replacing all matches in a diagram is repeated for each diagram.

The replace operation for each selected match is executed separately. The correspondence between the match and the find pattern is used for executing the replace operation. The execution of the replace operation relay on element types in a DSL. In general, the execution of

the replace step consists of the following substeps (some substeps may be skipped if they are not necessary):

- Create a target structure: The semantics of the replace pattern is to create all elements on the right-hand side of a find-replace edge.
- Copy attribute values: A value of an attribute in the source element is copied to an attribute in the target element if they have the same name and type. If there is no appropriate attribute in the target model then it is ignored. If elements at the source and target of the find-replace edge have the same type it is possible to copy all attribute values. If multiple find-replace edges enter a node attributes from all source elements of find-replace edges are copied. If multiple source elements have attributes with the same name and type the expressions should be used.
- Move edges to target: For each edge type, there is a collection of the node types usable as a source of the edge and node types usable as a target of an edge. Therefore, only edges usable with the target node type are moved to the new element. Other edges are ignored in the step. (They are removed afterwards.) If elements at the source and target of the find-replace edge have the same type it is possible to move all edges. If multiple find-replace edges enter a node the edge ends are moved from all source nodes of find-replace edges.
- Set attribute values using expressions defined in the target elements: The simple expression language is used to set attribute values. It is necessary to use expressions if names of attributes differ or if there are attributes with the same name in multiple elements. See expression example in Fig. 6. The following elements are supported in the expression language: Attribute name (supported if only one find replace-edge enters the target element), Name of the find-replace edge followed by the attribute name (the name of the find-replace edge is prefixed with the "@" symbol and followed with "." and the attribute name), Constants, String operations of previously mentioned elements (concatenation, substring, split etc.).
- Remove instances of source elements (related to the find-replace edge): All related elements e.g., edges not moved in the previous step are removed. All nodes at the source of the find-replace edge are removed. The edge is removed if it satisfies the following conditions: the edge is included in a match processed; both its ends have outgoing find-replace edges. A specific find-replace element type - delete node is used if it is required to remove a node.

4. Find-Replace in a Tool

It is possible to implement find-replace principles in any graphical DSL tool development framework. However, the encoding used by tools differ. An important part of find-replace transformation implementation is the translation of the concrete syntax into the abstract syntax. This part is tool-specific. Another important issue is the technical space supported by the tool.

We implement find-replace in the web-based DSL tool development framework - ajoo [5]. A DSL editor in the ajoo framework is created by defining diagram types and their node types and

edge types. For each edge type supported source and target node types are defined. For the node and edge types supported attribute types (named compartments types in the ajoo framework) are defined. There is a specialization relationship between node types. The semantics is that children inherit edge types supported by the parent. Diagrams in the ajoo framework are encoded similarly. Diagrams consist of nodes and edges. Nodes and edges contain attributes (compartments). Each instance is related to its type.

To enable find-replace in a DSL, we provide service to extend the definition of a DSL with find-replace specific elements. We extend DSL definition with the find-replace edge type, delete node type and also an abstract node type. The abstract node type is serving as the source and the target of the find-replace edge. All node types in the DSL diagram and the delete node type inherit from this abstract node type. This way the find-replace edge is usable between any two node types in a DSL diagram.

The ajoo framework is based on Meteor [11] and uses MongoDB [10] as a data store. We implemented find-replace from scratch as to our knowledge there is no model transformation approaches working in the Meteor - MongoDB technical space. However, if the technical space already has support for model transformations (for example, EMF-based framework Sirius), find and find-replace patterns may be translated to model transformation rules.

5. Find-Replace Usage Scenarios

We discuss a few scenarios where find-replace may be useful for a DSL user.

- Language evolution: As a graphical DSL evolves (the language changes) it is required to update DSL models accordingly. Find-replace may be used to update models. At first, the DSL is extended with new elements. Afterwards, find-replace is used to transform models to the new constructs. In the end, unnecessary DSL elements are removed.
- Model evolution: Find-replace may be useful to adapt a model according to changes required. In this case, the language is the same, but usage principles and conventions changes. For example, it may be necessary to rename model elements according to new naming conventions or to restructure diagrams.
- Model clean-up: Detect and remove harmful elements from the model. This may be necessary if some constructs cause problems in DSL execution. The find-replace may be used to modify problematic elements or to remove unnecessary elements. This approach may also be used to improve the readability of the models or to check whether models conform to naming and/or formatting conventions, etc.

We demonstrate the model evolution using simple activity diagram example. Activity diagram describing ticket vending process is given in Fig. 2. There are four node kinds in the diagram: *start symbol* (black circle), *end symbol* (black circle with white halo), *action* (rounded rectangle), *condition* (diamond). The action has two properties *performer* (black text) and *name* (blue text). Beside nodes, there are edges with optional *conditions*.

As with text documents, it is important to maintain the consistency of models, as well. We have identified three required improvements in the diagram. The first improvement required in the diagram is to modify the performer name of actions performed by the ticket vending

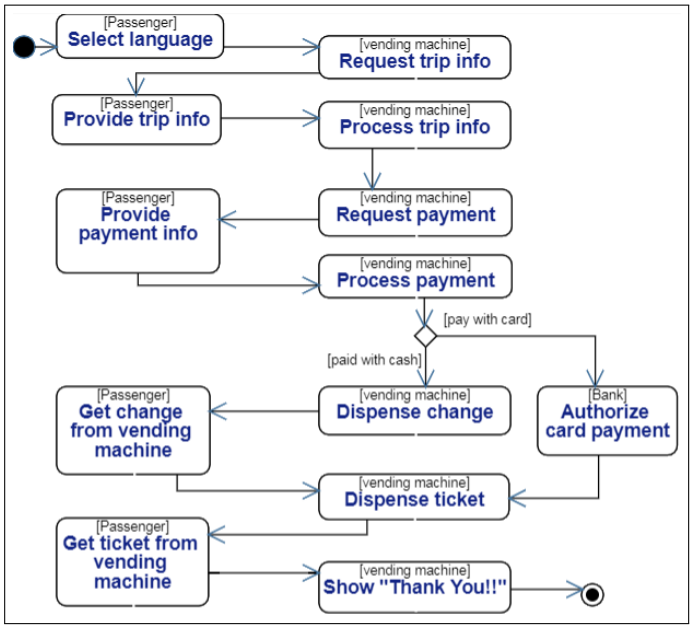

Figure 2: Example of activity diagram describing "Ticket vending process".

machine. It is required to replace the performer "vending machine" with the performer "Ticket vending machine". The find-replace transformation is given in Fig. 3. The result of the find query of this transformation (in the ajoo framework) is given in Fig. 4. It is possible to highlight find results in a diagram (Fig. 5.). Afterwards, it is possible to replace one match or all matches in a diagram. The result after replacing all matches in a diagram is given in Fig. 5.

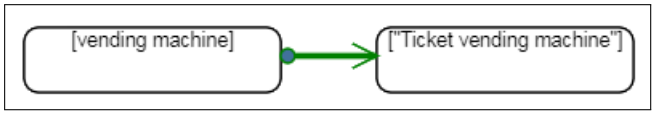

Figure 3: Find-replace diagram replacing performer "vending machine" with performer "Ticket vending machine". Implementation in the ajoo framework is shown.

The second improvement required is to introduce *merge symbol*. There are various styles of how transitions are used in activity diagrams. In the example (Fig. 2) multiple edges enter action node. There is another style, where only one transition can enter an action. To merge multiple flows *merge symbol* (diamond similar to *condition*) is used. Find-replace pattern introducing merge symbol and the result of the transformation is given in Fig. 6.

The third improvement of the example is to extract *Object* from actions. In the UML activity diagrams, there is *Object* symbol used to describe real-life objects sent from one action to another action. If the ticket vending machine "Dispense" something to a passenger, it should be extracted as an object in the activity diagram. The transformation and result is given in Fig. 6.

The find-replace patterns were demonstrated using one DSL diagram. However, the find step of a transformation finds matches in all diagrams accessible to a user.

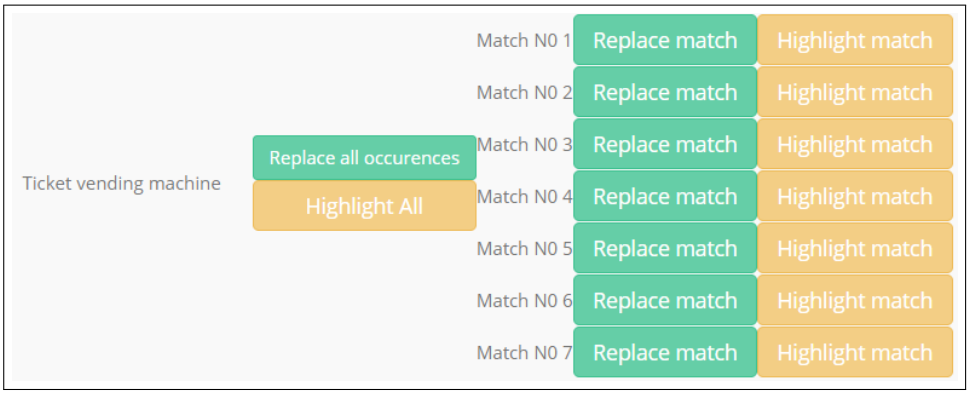

Figure 4: The result of the find-replace query given in Fig. 3, shown as implemented in the ajoo.

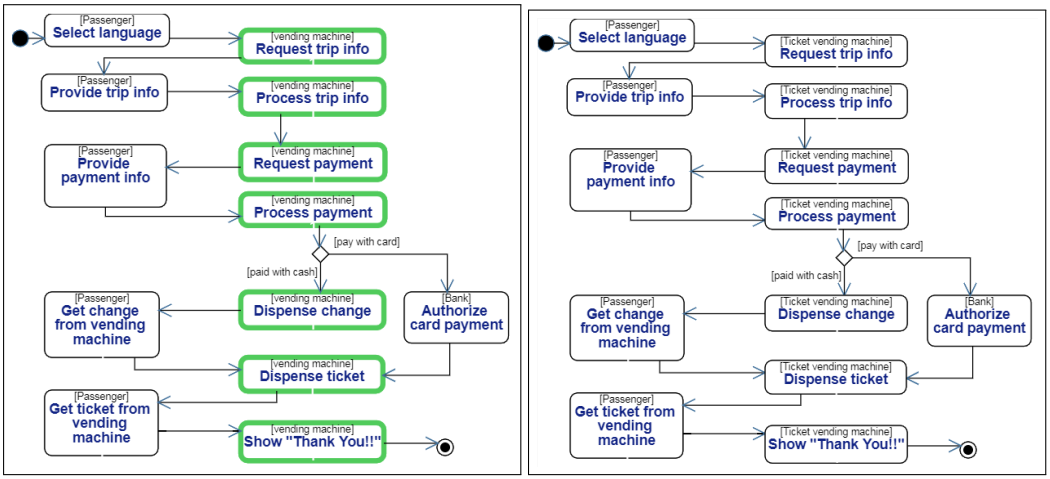

Figure 5: Results of the find-replace query (given in Fig. 3) highlighted in the "Ticket vending" example given in Fig. 2 are shown on the left. The highlighting is shown as implemented in the ajoo framework. The result of the replace all operation of the find-replace query is given on the right.

6. Related Work

The find-replace is a model transformation of a special kind. It is in-place transformation: endogenous, unidirectional, the same model is used as a source and as a target of a model transformation. There are additional requirements set for the find-replace task:

- It should be possible to execute transformation for all or some manually selected matches only. Therefore, it should be possible, to distinguish find step and replace step.
- The modeller working with a DSL should be able to define find - replace transformation, therefore the concrete syntax of the DSL should be used.
- The transformation should work in the technical space of a DSL tool development framework.

Traditionally, model transformation is defined using the abstract syntax of the language.

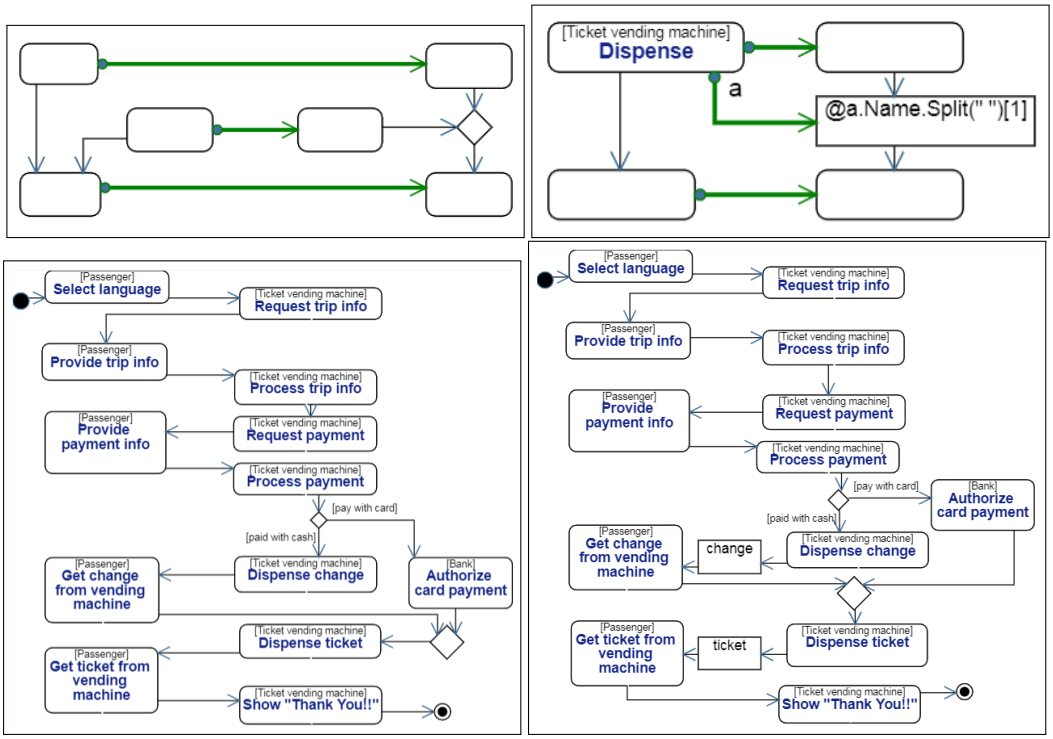

Figure 6: Find-replace transformation introducing *merge symbol* is shown on the left. Find-replace transformation extracting *object* is shown on the right. Result of transformation on example given in Fig. 5 are shown under the transformation.

However, there are concrete syntax-based approaches, as well. The concrete syntax-based approaches are: concrete syntax-based model transformations [12, 13, 14, 15, 6, 16], model transformations by example [17, 18, 19] and model transformations by demonstration [20, 21, 22].

In general, the concrete syntax-based find-replace is a subset of concrete syntax-based model transformations. Still, the find-replace use case sets certain constraints on concrete syntax transformation definition's means and style. There is no need for control structures because only a single find or find-replace pattern will be executed at a time. To ensure a similar style of find and find-replace features, the pattern used in the find query should be the same as the one used in the find-replace operations. It should be noted, that the find-replace is intended for small incremental updates of a model. It is not a full transformation language.

Source and target model pairs are used to generate model transformation in model transformations by example. It may be necessary to provide multiple model pairs, to provide correspondences or to edit generated transformation, depending on the approach. Model transformations by demonstration is a special type of model transformations by example. Here the target model is obtained using demonstrations. The demonstration-based approaches are endogenous, but they don't use correspondences. Model transformation by example and model transformation by demonstration approaches rely on constraint guessing for transformation. In our approach, the constraints are stated explicitly by setting attribute values. The [23] analyses the first wave

of concrete syntax-based approaches. It states that "no correspondence-based approach has been proposed for endogenous transformations, so far." To our knowledge, there is still no other endogenous correspondence and concrete syntax-based approach.

Model transformations are defined as annotated concrete syntax diagrams of a tool in VMTL [16]. Although these ideas may be generalized to DSL tool building frameworks, the current implementation requires defining a transformation from each language/tool used to the format supported by the transformation execution engine. However, in our approach, find-replace features should be available to any language defined in the DSL tool definition framework. It should also be noted that in VMTL a special language of annotations is used in the patterns.

Concrete syntax-based model transformations are integrated with the DSL tool development framework in the AToMPM [6]. The transformation rule in the AToMPM consists of Left - Hand Side, Right - Hand Side and Negative Application conditions. To relate Left - Hand Side and Right - Hand Side elements "__pLabel" property is used. We in contrast use a visual representation of relations - the find-replace edge. Constraint and action code in AToMPM patterns is written in Python. Our approach targets modellers, often without a programming background. We think that for users without programming background it is easier to gasp relations than variables. The AToMPM supports full-fledged transformation language while our approach is tailored for small incremental model updates. Besides, currently, it is not possible to execute AToMPM rules only for some manually selected matches as it is required in the find-replace use case.

7. Conclusions

In the paper, a universal find-replace approach as a service of DSL tool development framework is proposed. The find-replace operation is performed in two steps. At first, the find operation is performed. The replace operation is performed for each match of the find operation separately. The user may select to perform the replace operation for one or multiple matches at once. If there are conflicting matches, the user must select one to be replaced. The find-replace is implemented in the web-based DSL tool development framework ajoo [5]. However, the approach could be implemented in other DSL tool development frameworks, as well. The approach relies on element types. The encoding used by tools differ, but element types are present in one way or another in almost all DSL tool development frameworks. It should be noted, that other graphical element types are supported in other tools, like box in a box or pins. The approach proposed in the paper could be extended to support other element types, as well.

The find-replace approach proposed in the paper is not a complete concrete syntax-based transformation language. Thus, it is not possible to write any transformation as a find-replace pattern. It is not the purpose of the approach. The find-replace is intended for small incremental updates of a model. This is how find-replace is typically used in other tools (text editors, etc.).

Acknowledgments

The work has been supported by the European Regional Development Fund within the project # 1.1.1.2/16/I/001, application # 1.1.1.2/VIAA/1/16/180 “Concrete Syntax Based Find for Graphical Domain Specific Languages”.

References

[1] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, R. Heldal, Industrial adoption of model-driven engineering: Are the tools really the problem?, in: A. Moreira, B. Schätz, J. Gray, A. Vallecillo, P. Clarke (Eds.), Model-Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–17.

[2] MetaCase, Metaedit+ 5.1., 2020. URL: http://www.metacase.com/.

[3] Microsoft, Modeling sdk for visual studio - domain-specific languages., 2020. URL: https://msdn.microsoft.com/en-us/library/bb126259.aspx.

[4] Eclipse, Sirius, 2020. URL: https://www.eclipse.org/sirius/.

[5] A. Sprogis, Dsml tool building platform in web, in: G. Arnicans, V. Arnicane, J. Borzovs, L. Niedrite (Eds.), Databases and Information Systems, Springer International Publishing, Cham, 2016, pp. 99–109.

[6] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, Atompm: A web-based modeling environment., Demos/Posters/StudentResearch@ MoDELS 2013 (2013) 21–25.

[7] E. Kalnina, A. Sostaks, Towards concrete syntax based find for graphical domain specific languages, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2019, pp. 236–242. doi:10.1109/MODELS-C.2019.00038.

[8] E. Kalnina, Concrete syntax-based find for graphical dsls, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20, Association for Computing Machinery, New York, NY, USA, 2020. URL: https://doi.org/10.1145/3417990.3422008. doi:10.1145/3417990.3422008.

[9] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, Eclipse Series, 2 ed., Addison-Wesley, Upper Saddle River, NJ, 2009. URL: https://www.safaribooksonline.com/library/view/emf-eclipse-modeling/9780321331885/.

[10] I. MongoDB, Mongodb, 2020. URL: https://www.mongodb.org/.

[11] M. Software, Meteor, 2020. URL: https://www.meteor.com/.

[12] T. Baar, J. Whittle, On the usage of concrete syntax in model transformation rules, in: I. Virbitskaite, A. Voronkov (Eds.), Perspectives of Systems Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 84–97.

[13] M. Schmidt, Transformations of uml 2 models using concrete syntax patterns, in: N. Guelfi, D. Buchs (Eds.), Rapid Integration of Software Engineering Techniques, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 130–143.

[14] R. Grønmo, B. Møller-Pedersen, G. K. Olsen, Comparison of three model transformation languages, in: European Conference on Model Driven Architecture-Foundations and Applications, Springer, 2009, pp. 2–17.

[15] R. Grønmo, Using Concrete Syntax in Graph-based Model Transformations, Ph.D. thesis, University of Oslo, 2010. Doctoral thesis.

[16] V. Acretoaie, H. Störrle, D. Strüber, Vmtl: a language for end-user model transformation, Software & Systems Modeling 17 (2016) 1139–1167.

[17] M. Kessentini, H. Sahraoui, M. Boukadoum, O. B. Omar, Search-based model transformation

by example, Software & Systems Modeling 11 (2012) 209–226.
[18] D. Varró, Model transformation by example, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2006, pp. 410–424.
[19] M. Wimmer, M. Strommer, H. Kargl, G. Kramler, Towards model transformation generation by-example, in: 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07), IEEE, 2007, pp. 285b–285b.
[20] P. Langer, M. Wimmer, G. Kappel, Model-to-model transformations by demonstration, in: L. Tratt, M. Gogolla (Eds.), Theory and Practice of Model Transformations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 153–167.
[21] V. Acretoaie, H. Störrle, D. Strüber, Transparent model transformation: Turning your favourite model editor into a transformation tool, in: D. Kolovos, M. Wimmer (Eds.), Theory and Practice of Model Transformations, Springer International Publishing, Cham, 2015, pp. 121–130.
[22] Y. Sun, J. White, J. Gray, Model transformation by demonstration, in: A. Schürr, B. Selic (Eds.), Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 712–726.
[23] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Model transformation by-example: A survey of the first wave, in: A. Düsterhöft, M. Klettke, K.-D. Schewe (Eds.), Conceptual Modelling and Its Theoretical Foundations: Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 197–215. URL: https://doi.org/10.1007/978-3-642-28279-9_15. doi:10.1007/978-3-642-28279-9_15.