TEXTUAL SYNTAX MAPPING CAN ENABLE SYNTACTIC MERGING

László Angyal, László Lengyel, Tamás Mészáros and Hassan Charaf
Department of Automation and Applied Informatics, Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary

Keywords: DSL, Round-trip Engineering, Textual Syntax Mapping, Incremental Synchronization.

Abstract: As the support is increasing for textual domain-specific languages (DSL), the reconstruction of visual models from the generated textual artifacts has also come into focus. The state-of-the-art bidirectional approaches support reversible text generation from models using single syntax mapping. However, even these tools have not gone such far to facilitate the synchronization between models and generated artifacts. This paper presents the importance of synchronization and how these mappings can enable syntactic reconciliation for custom DSLs. Our approach provides algorithms for supporting incremental DSL-driven software development, which enables the freedom of choosing between the textual or visual editing of artifacts. It depends on the developer which representation is more effective for her/him at a specific moment.

1 INTRODUCTION

In the practice of model-based software development, the software models are usually represented as labeled (attributed), typed graphs. The modeling elements as nodes are connected to each other via edges. In modern modeling frameworks (Angyal et al, 2009) (Eclipse, 2010) (Xtext, 2010) visual and/or textual notations can be mapped to the nodes and edges of a metamodel to determine how its instance models should be drawn or written. These are referred to as the concrete syntax, which is required to define instance models.

The most prevalent techniques for textual syntax definition of modeling elements are originated in the theory of parser generators. The textual model can be parsed into an Abstract Syntax Tree (AST), which can be considered as an abstract model conforming to the AST metamodel. Every node in the metamodel have an AST class representation. The philosophy behind reversible text generation approaches is that parsing the textual artifact into an intermediate AST can be the input to recreate the model. However, this is inadequate to support the concurrent evolution of the visual and the textual representations of the same model. These approaches consist of two unidirectional synchronizations, often referred to as destructive, which means that the target model is not modified incrementally and thus, the update rebuilds the new content, instead of modifying the existing one.

Accordingly, the layout of the models that is previously defined by human effort disappears. The layout in both visual and textual representations contains valuable extra information, which is developed into the model. Only an incremental approach can preserve the layout, because the affected parts are updated only, while other parts remain unchanged.

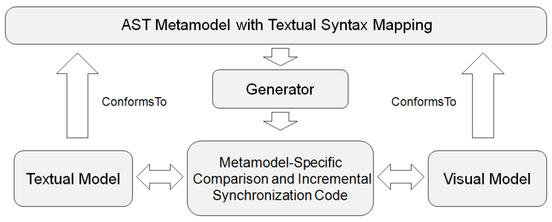
AST Metamodel with Textual Syntax Mapping
Generator
ConformsTo
ConformsTo
Textual Model
Metamodel-Specific Comparison and Incremental Synchronization Code
Visual Model

Figure 1: Outline of our approach.

We introduced a technique (Angyal et al, 2008) that performs a three-way AST comparison and incremental change propagation in order to reconcile the differences between a source code file and a visual model. Based on the syntax mapping, that synchronization approach can be extended to support arbitrary DSLs. In this paper, we demonstrate how the single syntax specification can

Angyal L., Lengyel L., Mészáros T. and Charaf H. (2010).
TEXTUAL SYNTAX MAPPING CAN ENABLE SYNTACTIC MERGING.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 308-311
DOI: 10.5220/0003011403080311

be applied to implement a metamodel-specific incremental merge approach. (Figure 1).

The remainder of the paper is organized as follows. The next section introduces the backgrounds. Section 3 contains an overview about our proposal for applying textual syntax mappings in an incremental synchronization approach of DSLs. Finally, conclusions are elaborated.

2 BACKGROUNDS

The initial and fundamental step in textual DSL development is the construction of the AST metamodel (the vocabulary) and the definition of the textual syntax for the elements. To exclude possible but illegal model states, constraints can be defined on the structure.

Processing textual models e.g. to generate other artifacts, requires them to be parsed and converted into a format, which is supported by a model processor or a generator. Formerly, for textual languages, a grammar file defined by developers was used, from which a parser generator (e.g. ANTLR, 2010) created a parser that could be integrated into a custom tool.

As the AST metamodel-based approaches have come into consideration in the language engineering researches, more and more tools and approaches are being developed to facilitate the definition of custom textual languages.

TCS (Jounault et al, 2006) is a textual DSL intended to bridge the modeling and the syntax worlds. From a *TCS* model, the grammar file for text-to-model transformation and text generator for model-to-text transformation can be produced.

Xtext (Xtext, 2010) and *MontiCore* (Krahn et al, 2007) are frameworks for development of textual DSLs. In order to reduce the redundancy of the metamodel and the concrete syntax, the definitions of the abstract and the concrete syntaxes for the languages are integrated into a single grammar file. Their generators produce a parser, an AST-metamodel as well as a full-featured text editor.

Although synchronization is a well-known problem in the practice of software development, the recent researches in the context of DSL engineering are still not focusing on it.

Coarse-grained file comparison approaches like the *diff* tool considers the lines as atomic building blocks. However, to compare two pieces of textual model correctly, the algorithm must take the grammar of the language into consideration. The fine-grained algorithms operate on the ASTs of the source code files. Hierarchical structures such as models should also be treated as source code.

3 THE MERGE APPROACH

3.1 Bidirectional Textual Syntax Mapping

The metamodel itself does not determinate how its instance models should be drawn or written. For a complete language, the concrete syntax with the assignments to the metamodel is inevitable. Figure 2 depicts our meta-metamodel, where the *Template* attribute holds the textual concrete syntax mapping belonging to that node.

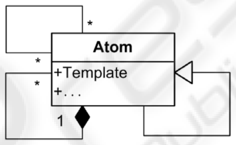

Figure 2: The meta-metamodel.

The *Atom* node represents the self-describing metatype for all elements in the models. An element can (i) define own, (ii) inherit attributes and relationships from its ancestor (inheritance), (iii) structures can also be defined: an element can contain other elements (containment), and (iv) other existing elements can be referenced (cross-reference), as well. Furthermore, the edges have multiplicity properties, which are taken into account in the parser and the text generator.

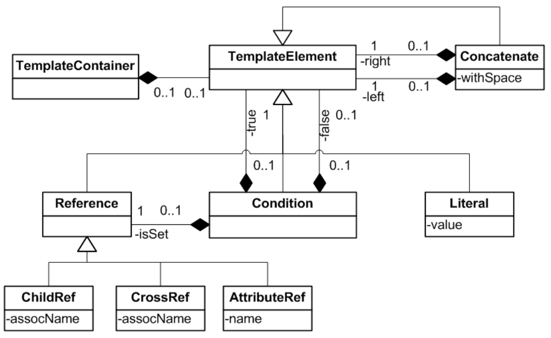

Figure 3: The metamodel of the template language.

We designed a simple template language (Figure 3) that expresses the textual appearance of abstract elements. These templates are considered as input artifacts for the template compiler, which produces the text generator and rules for the text parser.

The abstract *TemplateElement* is a word in that textual template, which can be a reference, a string literal or a condition. The words are concatenated into a template with the *Concatenate* operator. Additionally, each attribute has a data type with predefined regular expression (a primitive template), which can be overridden to determine the values allowed in that attribute. This regular expression is used by the parser to recognize the attributes.

3.2 Realizing the Incremental Update

3.2.1 Edit Scripts

A merge approach can be operation-based or state-based (Mens, 2002). The operation-based one requires recording the committed edit operations, while the state-based one derives the changes after they occur by a comparison. The sequence of these operations is referred to as an edit script. Our change propagation approach executes the edit scripts on other artifacts to obtain the same state.

3.2.2 Update Visual Model

The structure of our proposed incremental update component for the visual model is depicted in Figure 4. Since the underlying data types and classes are metamodel-specific, all of these components are generated and operate only on a specific model. Furthermore, these classes are grouped into larger logical units.

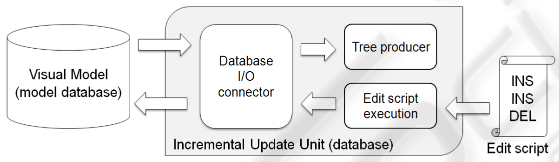

Figure 4: The incremental model update unit.

The *Database I/O connector* provides access to the model elements stored either in a database or in a file. The *Tree producer* reads the model from the database and produces the AST in a form required for the difference analysis. The *Edit script execution* submodule performs the incremental database update controlled by an edit script.

3.2.3 Update Textual Model

The component for the incremental textual update (Figure 5) includes the layout preserving logic. It contains a tree producer (denoted by *Parser*), which stores *trace information* linked to the AST nodes to facilitate the restoration of the original layout and comments. The *AST patch* module executes the edit script obtained from outside. After the incremental update, the reworked AST is pretty-printed considering the trace information and the original textual content.

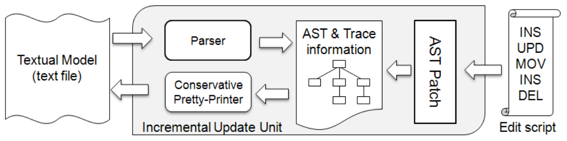

Figure 5: The incremental textual update unit.

3.2.4 Retaining the Textual Layout

The layout preservation is a crucial requirement to artifact regeneration: overwriting a customized layout with a generated one could be unacceptable for the developers.

The edit operations manipulate directly the AST, but while the reworked AST nodes or subtrees are printed, the trace links are taken into account to restore the original layout with comments in their original positions. Following the approach of (Fritzson et al, 2008), every operation in the edit script is converted into an equivalent text manipulation operation, which is applied on the original text file:

- Insertion: the new node is pretty-printed and its text is inserted into the text stream.
- Deletion: characters belonging to that node are removed.
- Update: a substring is replaced.
- Move: remove a substring and insert into an other position.

3.3 Composing the Techniques Together into a Sync Engine

Figure 6 illustrates our proposed synchronization engine (SE) with the three input models: M_0, M_1, and M_2. SE realizes an incremental three-way differencing-based merge, where the two modified artifacts (M_1, M_2) are compared to the last synchronized state (M_0) in order to unambiguously detect and propagate the committed refinements. The synchronization is performed with the help of intermediate artifacts, the ASTs.

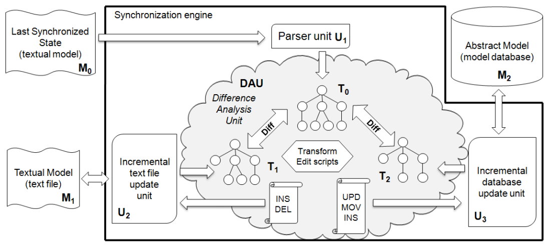

Figure 6: The synchronization engine (SE).

SE can be separated into four independent parts: (i) the difference analysis unit (*DAU*), which can be considered the heart of the SE, (ii) a simple parser unit (U_1), (iii) the complex incremental textual update unit (U_2), and (iv) the visual model manipulation unit (U_3). The last mentioned three units (U_1, U_2, and U_3) serve as language-specific glue units for the metamodel independent *DAU*.

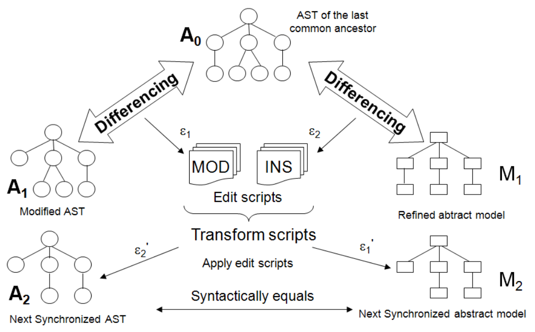

Figure 7: The synchronization approach in the details.

The algorithms in the *DAU* (Figure 7) operate on the ASTs. The procedures denoted by *Diff* based on the algorithms elaborated in (Chawathe et al, 1996) and have been customized for source code (AST) difference analysis. A general tree matching algorithm tries to find the correspondence between the two trees; the nodes that remained unmatched compose the differences. The edit scripts can reconcile the differences between the trees.

The modifications in the two artifacts can affect each other, since the nodes are identified by their path. An inserted node can shift the indices and may cause that an edit operation from the opposite edit script can address a different node. To avoid this, we transform (by incrementing or decrementing the indices) the paths in the operations if they affect each other. Finally, to propagate the changes, the transformed edit scripts are executed on the other side by the generated glue units (U_2, and U_3). At the end of the synchronization we obtain two syntactically equivalent artifacts.

4 CONCLUSIONS

The accelerated spread of the DSLs requires the development of tools to support the evolution of both the visual and textual languages. This means just the beginning towards the round-trip engineering and incremental synchronization between independently, and concurrently evolved DSL models.

The presented synchronization technique involves structural syntactic model-text differencing and three-way AST merging. The main advantage is that in contrast to typical text generation approaches, it permits modifying the generated textual artifacts and instead of losing the changes, they will be synchronized back to the models. The modular design allows the model-model and in addition the text-text synchronization. On models where semantic conflicts never occur, this approach can be used efficiently.

ACKNOWLEDGEMENTS

This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

REFERENCES

Angyal, L., Lengyel L., Charaf, H. 2008. Novel Techniques for Model-Code Synchronization. In *Proceedings of The 3rd International ERCIM Workshop on Software Evolution*. Electronic Communication of the EASST, 8.

Angyal, L., Asztalos, M., Lengyel, L., Levendovszky, T., Madari, I., Mezei, G., Mészáros, T., Siroki, L., Vajk, T., 2009. Towards a Fast, Efficient and Customizable Domain-Specific Modeling Framework. In *Proceedings of the IASTED International Conference*. Innsbruck, Austria.

ANTLR, 2010. ANTLR Parser Generator, http://www.antlr.org

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., Widom, J., 1996. Change detection in hierarchically structured information. In: *Proceedings of International Conference on Management of Data*, Montreal, Canada, pp. 493-504.

Eclipse EMF, 2010. http://www.eclipse.org/emf

Fritzson, P., Pop, A., Norling, K., Blom, M., 2008. Comment- and Indentation Preserving Refactoring and Unparsing for Modelica. In: *Proceedings of 6th International Modelica Conference*, Bielefeld, Germany, pp. 657-666.

Jouault, F., Bezivin, J., Kurtev, I., 2006. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: *Proceedings of Generative programming and component engineering*. Portland, USA, pp. 249-254.

Krahn, H., Rumpe, B., Völkel, S., 2007. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: *Model Driven Engineering Languages and Systems* (4735), pp. 286-300.

Mens, T., 2002. A State-of-the-Art Survey on Software Merging, In: *IEEE Transactions on Software Engineering*. 28(5), pp. 449-462.

Xtext, 2010. http://www.eclipse.org/Xtext