Electronic Communications of the EASST
Volume 36 (2010)

Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010)

Support for Bidirectional Model-to-Text Transformations

Anthony Anjorin, Marius Lauder, Michael Schlereth and Andy Schürr

15 pages

Guest Editors: Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Support for Bidirectional Model-to-Text Transformations

Anthony Anjorin[1]*, Marius Lauder[2]†, Michael Schlereth[3] and Andy Schürr[4]

[1] anthony.anjorin@es.tu-darmstadt.de
[2] marius.lauder@es.tu-darmstadt.de
[4] andy.schuerr@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de
Technische Universität Darmstadt, Real-Time Systems Lab,
Merckstr. 25, 64283 Darmstadt, Germany

[3] michael.schlereth@siemens.com
Industry Sector, Industry Automation & Drive Technologies Divisions,
I IA&DT ATS 41, Siemens AG,
Gliewitzerstraße 555, 90475 Nuremberg, Germany

Abstract: In recent years, model-driven approaches and processes have established themselves as pragmatic and feasible solutions with tangible advantages. Transformations play a central role in any model-driven solution and, as interest in *textual modelling* grows, providing concepts and tools for supporting a high-level and declarative specification of *bidirectional model-to-text transformations* becomes a vital area of research. Our paper identifies important areas and scenarios for model-to-text transformations that are not or only partially supported by currently existing solutions. Based on the requirements of a real-world case study, we introduce a new concept that has been inspired by a successful bidirectional model-to-model transformation approach: Triple Graph Grammars.

Keywords: textual modelling, model-to-text transformations, bidirectional transformations, pair grammars, triple graph grammars, AST

1 Introduction and Motivation

A model-driven approach requires the right abstraction or *model* for a certain task and establishes an appropriate *Domain Specific Language* (DSL), with which solutions can be expressed in a high-level manner. A corresponding model-driven process or solution promises a series of advantages including an improvement of software quality and better support for reusability. For these reasons and many more, the model-driven paradigm is establishing itself as a realistic and practical way of designing and improving software systems with a short turnaround time and tangible advantages [SV05].

* Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.
† Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

Transformations: model-to-model and model-to-text, unidirectional or bidirectional, play a central role in successfully establishing a model-driven solution. In this paper, we focus on (bidirectional) model-to-text transformations, which are of great relevance in the context of textual modelling.

Based on the requirements of a real-world case study, our contribution shows scenarios that are not or only partially supported by existing approaches and solutions. These include ignoring large text fragments in a text-to-model transformation, catering for complex real-world programming languages that require (partially) manually programmed parsers, and providing *explicit* support for traceability links and bidirectionality. As an answer to these requirements, we present a novel bidirectional model-to-text transformation language inspired by so-called Triple Graph Grammars (TGGs) [Sch94].

This paper is structured as follows: Section 2 introduces our case study, showcasing pertinent requirements which are further discussed in Sec. 3, which considers state-of-the-art solutions, identifies open problems, and presents preliminary results that culminate in our current vision for a bidirectional model-to-text transformation language. This is informally introduced in Sec. 4 based on the requirements of the case study. Section 5 concludes with a brief summary and an outlook on future work.

2 Case Study

In the last two years, we investigated the field of Model-Driven Automation Engineering (MDAE) [SRS09, LSRS10] in cooperation with Siemens AG. Automation engineering is a complex discipline where experts of *different* domains work in *parallel* to set up highly complex machine systems. As an example of such a system, we decided to explore a high-bay warehouse storage and retrieval system that consists of many components such as motion control units, communication devices, and drive units. The different domain experts involved in the development process typically use their own well established and trusted *domain specific* workflows, processes, and tools.

To reduce the complexity of the case study, we focused on two typical types of information in different domains: A set of commands in *Statement List* (STL)[1] in automation engineering, and a *Location Oriented Structure* (LOS)[2] in electrical engineering. In this case study, we provide engineers with a new tool that enables them to synchronise data created and/or modified using their own different and established tools. Setting up such a *tool integration* requires close collaboration with domain experts to identify relevant aspects of each tool in a corresponding metamodel[3], and to define a set of correspondences as traceability links that can be used to ensure consistency. Keeping a program in STL consistent with an LOS model requires a bidirectional model-to-text transformation as STL code must be parsed (text-to-model) and LOS models *unparsed* (model-to-text).

As depicted in Fig. 1, the engineering tools involved are *Comos ET* (Fig. 1::1) for editing LOS models, and *SIMATIC STEP7* (Fig. 1::2) for editing STL code. To integrate these different

[1] STL in SIMATIC STEP7 corresponds to the “Instruction List” language defined in IEC 61131-3.
[2] Defined in IEC 61346-1.
[3] A model that describes other models.

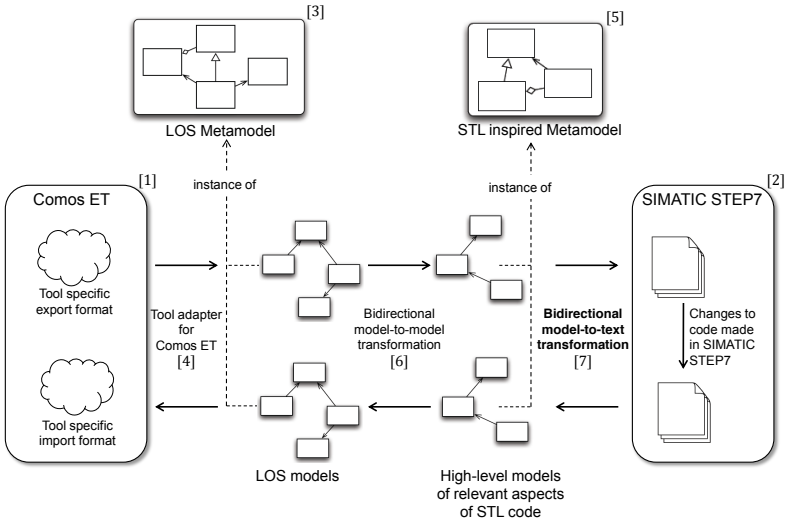

Figure 1: Workflow and central components of our tool integration case study.

tools, an LOS metamodel (Fig. 1::3) was specified and a *tool adapter* (Fig. 1::4), to transform from the export/import format used by Comos ET to instances of the LOS metamodel, was implemented. An *STL inspired metamodel* (Fig. 1::5), representing relevant aspects of STL programs, was also established, with which the LOS $\leftrightarrow$ STL integration could be implemented via a bidirectional model-to-model transformation using TGGs (Fig. 1::6), and a bidirectional model-to-text transformation (Fig. 1::7) from STL code models to corresponding STL programs. The use of an appropriate STL inspired metamodel (Fig. 1::5), instead of directly mapping LOS models to text and vice-versa, has two advantages: firstly, the model-to-text transformation is simplified, as the STL inspired metamodel is conceptually closer to the textual syntax to be handled, and secondly, the established model-to-text transformation can be reused in a totally different context.

It is important to note that STL is neither a textual view nor a *textual concrete syntax* for an STL model. STL is a programming language used to specify the tasks to be run on a *SIMATIC Programmable Logic Controller* (PLC) and is defined by Siemens AG. In our context, the STL inspired metamodel is *intentionally* incomplete and can be regarded as an abstraction of a complex programming language, suitable for the specific integration. This metamodel is the result of intense collaboration with domain experts and has been extended and refined in an iterative manner.

Figure 2 depicts a simplified version of the STL inspired metamodel for our case study. A program in STL consists of files that may either be organisational blocks (OBs) or functions (FCs). An FC file contains concrete commands to control and/or use certain devices or device items that are to be programmed, e.g. the movement of a motion unit dependent on the output of some sensor. In an OB file, different environment variables are set and the execution order of the FCs is coordinated. An FC can be *used* or invoked from an OB via *CALL commands* in the OB file. These call commands are, for example, of the form `CALL FC 32;` where 32 is the *address* of some FC. As a first iteration, we focused only on the usage of FCs by an OB, which has a title

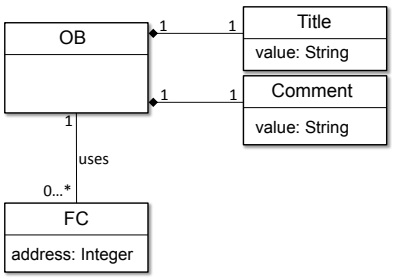

Figure 2: A metamodel for a small excerpt of STL, relevant for our integration.

and a comment. All other details of STL are left out and handled as static text fragments.

The other data structure to be integrated is described by the LOS metamodel (Fig. 1::3) that specifies the structure of a machine system from a more physical point of view. The system is classified starting from the installation location and along its placement relationships (e.g. a device is placed *inside* a control cabinet). As our focus in this paper is on the required bidirectional model-to-text transformation, we will not go into further details on the LOS metamodel.

LOS models and corresponding STL code have points of tangency, where information is represented in both forms. As an example, consider a certain drive device in an LOS model, representing an axis of a storage and retrieval machine of the high-bay warehouse system, which is programmed via an FC file in STL. Such explicit coupling enables the exchange of information in a bidirectional manner. The goal is to support engineers within their established workflows by providing a powerful tool that enables fast information exchange and consistency checks. Once again, as we focus on the necessary model-to-text transformation, the integration model for the case study is not discussed further (cf. [LSRS10] for details).

The presented case study requires a tool integration technique that supports the following requirements:

- Information has to be exchanged in both directions, from a specific STL program to a specific LOS model and vice-versa. Therefore, *bidirectional* transformations are necessary.

- Established workflows using standard engineering tools (e.g. Comos ET for LOS and SIMATIC STEP7 for STL) have to be supported and complemented and cannot be changed by enforcing the use of a new and different tool or editor. Therefore, a technique has to be developed that enables us to use the data as provided by the corresponding tools.

- Clear and simple interfaces for integrating and using possibly preexisting parsers and metamodels built with standard tools (parser generators, CASE-tools).

- The integration with a standard programming language requires an iterative process of abstracting from irrelevant parts of the textual syntax and focusing on aspects that are relevant for the integration. In our experience, this corresponds to how domain knowledge is obtained in numerous discussions and meetings with domain experts.

3 Related Work

State-of-the-art (bidirectional) model-to-text transformations can be grouped according to different criteria. An important and very general classification is achieved by making a clear distinction between approaches for supporting *concrete textual syntaxes* for new DSLs, and approaches for dealing with arbitrary and often preexisting fixed textual (programming) languages.

3.1 Support for defining a new concrete textual syntax

When choosing a textual concrete syntax for a Domain Specific Language (DSL), one usually has the freedom to define the textual representation from scratch – possibly inspired by a pre-existing domain specific notation. For lightweight DSLs, it is furthermore vital to be able to implement the textual concrete syntax and appropriate tool support, as fast and as easily as possible. For a further detailed classification and evaluation of approaches that fall into this group, please refer to [GBU10].

Current challenges in this group include automatically generating tool support, supporting incremental updates, and providing an intuitive means of specifying the textual syntax, from which a parser, possibly an unparser (if bidirectionality is to be supported), and tool support like editors, outlines etc. can be generated. A viable solution ([GBU09], [Nag96]) is to provide a syntax driven editor, that treats the textual syntax as a *view* of the actual model. Furthermore, many approaches are grammar-based ([Nag96], [EV06], [KRV08]) and require the user to specify the textual syntax using usually an EBNF-like grammar, from which a metamodel, parser, editor etc. are generated.

3.2 Support for arbitrary pre-existing textual languages

The second group of model-to-text transformation approaches – according to our general classification – aspires to cater for arbitrary textual languages and does not assume a direct 1:1 correspondence with a model. Moreover, the syntax of the targeted textual language is usually fixed and cannot be changed at will. Members of this group include mostly unidirectional approaches such as parser generators (ANTLR [Par07]), and template languages (Xpand [EK06]). In this group, bidirectional approaches [BGSZ10] are usually restricted to a specific metamodel and are not meant as a general framework for specifying arbitrary model-to-text transformations.

Challenges here include explicit support for bidirectionality and traceability links, and a high-level, intuitive and homogeneous language for specifying the actual transformations in both directions simultaneously.

3.3 Identified challenges and open problems

Based on our case study (Sec. 2) and the experience gained working with our research partner (Siemens AG), we were able to identify the following points that are indeed relevant for a wide range of scenarios including tool-integration, and are not – or only partially – supported by state-of-the-art solutions:

3.3.1 Enabling an iterative process:

Handling parts of the textual syntax that are irrelevant for the current model is an important prerequisite for an iterative process. In our experience, domain knowledge about the textual syntax is constantly extended and corrected in several meetings with clients and domain experts. This is only possible if one is able to focus on a few aspects of the textual syntax at a time and treat the rest as static textual fragments that have to be present, but are currently irrelevant for model creation.

3.3.2 Working with arbitrary parsers:

Preexisting textual languages are usually already in use in a company or by a certain tool. This implies that parsers probably already exist for the language and should not be reimplemented. In some cases, the nature of the language (e.g. syntactical extensibility) requires a specific kind of possibly hand-written parser. It is, therefore, sometimes problematical to require a certain grammar and generate the corresponding parser automatically. A general solution must provide a simple interface for connecting to arbitrary parsers, and not aspire to replace powerful and stable parser generators.

3.3.3 Not enforcing a certain editor:

As is clearly the case in our case study based on SIMATIC STEP7 – it is often simply impossible to impose a certain editor. In the context of tool-integration (Fig. 1), a major goal is to be able to deal with changes *made using the corresponding tools*. Solutions that rely on specific features of syntax driven editors are therefore not generally viable in the case.

3.3.4 Ensuring replaceable templates:

We propose keeping templates as simple as possible to ensure that they remain a clean, replaceable view of the AST of the textual syntax [Par04].

3.3.5 Providing explicit support for bidirectionality:

Almost all approaches do not explicitly support bidirectionality and require the user to completely switch tools/languages. We propose providing support directly in the transformation language and argue that bidirectionality is often a necessity (such as in our case study) and is – in the majority of cases – always advantageous (e.g. for traceability links, handling manually edited code, etc).

3.3.6 Providing an intuitive, high-level transformation language:

A major problem with many code generators is that the resulting system is on a low level of abstraction and is hard to maintain, retarget or reuse in a different context. A solution would be to use a high-level declarative transformation language. The possibility and advantages of using *Triple Graph Grammars* (TGGs) to implement a bidirectional model-to-text transformation has

been considered and argued by [Wag09]. However, as a concrete implementation of a necessary parser and unparser is not discussed, we propose a solution that is *inspired* by TGGs but explicitly caters for the necessary interaction of all components (parser, templates and metamodel).

3.4 Preparatory Work and Results

In the context of our research cooperation with Siemens AG, an initial prototype was implemented for our case study. We chose a grammar-based approach, generating a parser, templates and a rudimentary metamodel from a *code specification* in simple EBNF with additional layout information. Our prototype supports *fully bidirectional* model-to-text transformations but is restricted to languages that can be expressed by our grammar. The lessons that we learnt and the experience gained include focusing on the actual transformation and interaction of standard parsers, user-defined templates and metamodels, rather than automatically generating the necessary components. Further work on an ANTLR based solution, leveraging ANTLR's *tree grammars* for abstracting from a parse tree, has led to our final vision – discussed in the following chapter – which uses pattern matching to allow for a declarative (what and not how) specification. Furthermore, it addresses all of the challenges identified in Sec. 3.

4 Code Adapters with Pair Grammars

Concepts in metamodelling in general and model transformations in particular, can be formalised, using the well researched and formally founded field of graph theory. As models consist of objects (nodes) and links (edges), the terms *model* and *graph* can be used synonymously. Along similar lines, model transformations can be formalised using *graph grammars* [EGL+05].

Triple Graph Grammars (TGGs), introduced by Schürr in 1994 [Sch94] were inspired by Pratt's *Pair Grammars*, originally presented in 1971 [Pra71]. The motivation for TGGs was to establish a declarative, intuitive and bidirectional model-to-model transformation language. Two models are coupled explicitly via correspondence links, by describing the coupling on the meta-level using TGGs to integrate the metamodels of the two modelling languages, being considered. On this meta-level, a set of declarative rules is also defined that express under which conditions elements from the two models are to be associated and how to create the appropriate correspondence links. TGGs describe the *simultaneous* evolution of three models (source, target and correspondence), which, although compact and intuitive to specify, cannot be used in practice. For this reason, different *transformators* are derived from a TGG that implement forward and backward transformations, consistency checks, and correspondence link creation. These transformators handle the necessary pattern matching and rule application conditions, as well as object creation. The reader is referred to [SK08] for the latest state-of-the-art TGG theory.

The advantages of a model-to-text transformation language inspired by TGGs include explicit support for bidirectionality and traceability links, a high-level intuitive specification of the transformation [Wag09], and a *seamless* user experience for continuing with our bidirectional model-to-model transformation language (TGGs). Our approach reflects back on the original idea and motivation for using pair grammars, namely, coupling a textual syntax with a corresponding graph grammar (metamodel).

MOFLON is a meta-case tool developed at the real-time lab at TU Darmstadt and supports, amongst other things, setting up a *tool integration environment* [KRS09]. To communicate with the tools involved, *tool adapters* are partly generated and have to be implemented to enable communication with the corresponding tool, either online via a tool API, or offline via an import/export of a tool-specific persistence format. We regard our pair grammar solution as a means of implementing a *code adapter*, thereby treating a certain textual syntax as just another tool-specific format in the context of a tool integration. Further applications include code generation and model extraction for arbitrary models.

With our extension, MOFLON would not only be a framework for metamodelling and specifying bidirectional model-to-model transformations, but also a flexible and homogeneous framework for specifying bidirectional model-to-*text* transformations and thus code adapters, code generators, or model extractors depending on the concrete use case.

4.1 MOCA (MOFLON Code Adapter Framework)

Figure 3 depicts all components and the data flow in our envisioned framework. The central component here is a *pair grammar* (Fig. 3::1) that specifies the connection between instances (Fig. 3::2) of the user-supplied metamodel (Fig. 3::3) and relevant parts of the text in an AST (Fig. 3::4). In the following Sec. 4.2, we will discuss the declarative pair grammar for our case study.

When going from model to text, the AST is generated for a corresponding model according to the rules in the pair grammar and transformed into text by an unparser (Fig. 3::5). The textual representation of an AST is unique as the tree structure can always be traversed in the same way (e.g depth-first). Specifying the textual representation for the AST is an iterative process, as the user starts with a default textual representation, and can add, change or replace templates to *decorate* nodes in the AST and thereby structure and extend the textual output as required. This set of templates (Fig. 3::6) defines the static parts of the textual syntax that were ignored by the parser and should be added for a complete result. Fig. 3::7 represents the textual syntax generated by the unparser and may consist of an arbitrary number of files.

For the text to model transformation, this *code* is passed on to the parser (Fig. 3::8) that decides which parts of the text are relevant and which not. This abstraction results in a concise Abstract Syntax Tree (Fig. 3::4) that is most suitable for the current transformation. The parser is an important component that the user must provide if a text-to-model transformation is required – for example using a parser generator or other means. The interface to our framework is a simple AST structure that the supplied parser must adhere to. It is important to note that the parser must *focus* on relevant parts of the text and should be iteratively extended to include more and more aspects as they become necessary for the transformation.

In our opinion, this is a flexible, powerful framework that allows the user maximum freedom. In our experience, the parser should usually be lenient and error tolerant with respect to the input text, while the unparser should always produce well formatted and structured code. If the user decides to retain white space or manually edited code and formatting, the parser can be implemented to be white space sensitive and add the necessary text fragments as nodes in the AST.

This means that one must distinguish between:

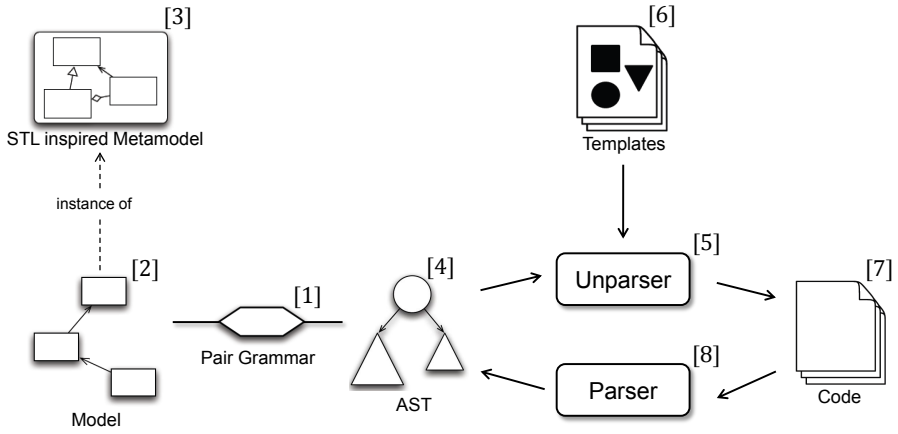

Figure 3: MOCA: MOFLON Code Adapter Framework

- static text fragments that are not to be manually edited and can be ignored by the parser and subsequently added when generating code via templates,
- and text fragments that are currently irrelevant for model extraction but can be manually edited/formatted by the user and should be preserved.

The latter cannot be completely ignored by the parser but should be added without further analysis as nodes in the AST. These can then be added to the model as simple attributes[4] with the corresponding textual content and written out unchanged when generating code. For simple textual syntaxes, [BGSZ10] shows that a rudimentary parser can be generated directly from the specified templates and does not need to be implemented. Our framework fulfils all identified requirements in Sec. 3.3, and the focus is clearly on the specification of the transformation (Fig. 3::1).

4.2 Pair grammar for our case study

A pair grammar consists of a set of rules. Each rule specifies how an AST fragment and a model fragment are to be built in parallel. Analogous to TGG theory, a set of forward and backward transformators can be derived from the rules that transform a model to an AST and vice-versa. As the right part of every rule is always an *instance of* the same simple AST-metamodel, we propose to simplify the specification by only requiring rules directly at the model level and deriving the necessary rule schemas at the metamodel level automatically. This is possible, because the AST metamodel is fixed and only contains simple concepts of a folder, a file and generic nodes with single values.

Figure 4 shows the three rules necessary for a simplified version of our case study. The left part of the rule is a valid fragment of an instance of the user supplied metamodel. Only elements and associations compatible to the metamodel can be used. Elements annotated with a "++" are

[4] With sensible default values.

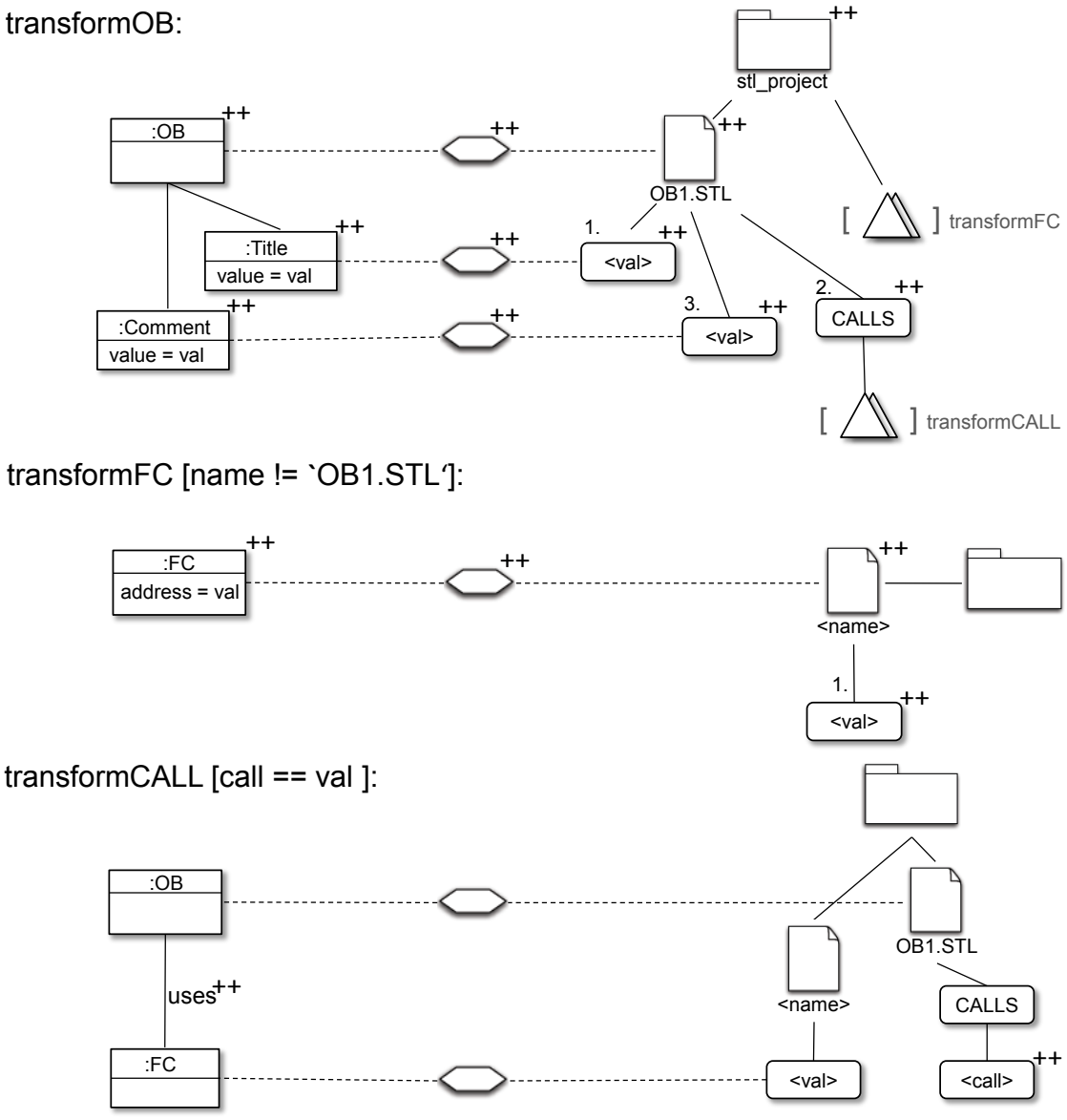

Figure 4: Pair grammar for our case study

to be created by the rule and do not need to be present for the rule to *match*. Elements without a "++" form the *context* of the rule and must be present for the rule to match[5].

The right part of the first rule `transformOB` describes the overall structure of an STL project. A folder (`stl_project`) contains a single OB file (`OB1.STL`) and multiple FC files. The OB file, as a file node in the AST, contains three *ordered* children: (1.) a node representing the title, (2.) a node `'CALLS'` containing all function calls as children, and (3.) a node representing a comment. Nodes in the AST are either folders or files with names, or generic nodes with a single value. The value can be explicitly specified (e.g. `CALLS`) or left variable (e.g. `<val>`). The parts of the rule in square brackets are not part of the grammar and just indicate that the subtrees, if present, would match the subrules `transformFC` and `transformCALL`.

The links between elements on different sides of the rules are so called *traceability links* and constitute a correspondence model that can be used to transfer attribute values, and specify further rules. This first rule `transformOB` consists only of elements marked with a "++" and, therefore, has no prerequisites or *context* and can always be matched and executed. The rule decides what model elements and nodes should be created by interpreting the corresponding structure and order of children of nodes in the AST.

The second rule in Fig. 4 requires that a folder already exists (folder on the far right doesn't have a "++" and thus belongs to the context of the rule). If an existing folder is found (e.g. after executing `transformOB`), every file whose name is *not* "OB1.STL" is linked to an FC element with the appropriate address. The additional constraint is specified as a condition to the rule and must be fulfilled for the pattern to match (`transformFC[name != 'OB1.STL']`). The distinction between context and non-context elements in a rule allows for a declarative and high-level specification of *what* and not *how* – and can be implemented with a pattern matching engine [GSR05].

The last rule `transformCALL`, is a very interesting rule with a relatively complex context. This rule *enriches* the model by searching for certain patterns in the AST. In this case, organisational blocks (OBs) *use* or call up functional blocks (FCs) if a corresponding call statement that matches the address of the FC is found in the OB. This is expressed by the pattern on the right part of the rule: a folder containing the OB and an arbitrary file are required. Both elements should already be linked to model elements on the left part of the rule (already translated). Non-context elements with a "++" specify that a "uses" association between an OB and an FC exists when a "call" node is found in the OB file, which matches the name of an existing FC file with the appropriate address (constraint is expressed again by the condition `transformCALL[call == val]`).

4.3 Application of our pair grammar using an example:

In this section, we consider an instance of the STL inspired metamodel (Fig. 1) as our input model and go through the steps of transforming it to STL code (model-to-text)[6]. The different rules from Fig. 4 have to be applied to the input model. To this end, a so-called *forward graph transformator* is derived from all rules, that expects all elements on the left of a rule to exist already (i.e. context objects) and creates the appropriate elements on the right and the links

[5] To increase readability, links in Fig. 4 are not annotated with "++" but of course, also have to be created.
[6] Text-to-model is analogous and will not be discussed in detail.

in between[7]. Figure 5 depicts the graph triple after applying the `transformOB` rule on the

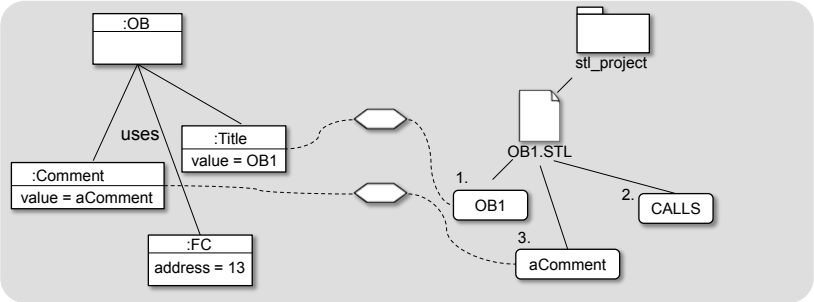

Figure 5: Model-to-text transformation after application of rule `transformOB`

model on the left. The result is the creation of a correspondence graph with its traceability links (centre of Fig. 5) as well as an AST with a folder `stl_project` and a file `OB1.STL` (right part of Fig. 5). This file has child nodes (1.) `OB1`, that represents its title, and (3.) `aComment`, representing a comment in the file. The child node (2.) `CALLS` is a structural node in the AST that indicates that all child nodes thereof are "CALLs" in the `OB` file.

The next rule that can match is `transformFC`. Applying it to the input model creates an `FC` file with a child node (1.) `13` and a link between the newly created `FC` file and its corresponding `FC` object (Fig. 6).

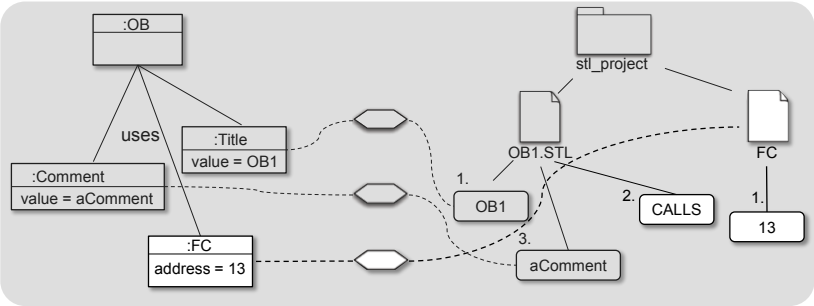

Figure 6: Model-to-text transformation after application of rule `transformFC`

The "uses" connection between the `FC` object and the `OB` object now has to be established in the AST. This is achieved by applying the `transformCALL` rule to the graph triple of Fig. 6. The result is depicted in Fig. 7. A new child node (1.) `13` is added to the node `CALLS`, and represents the corresponding call command in the `OB` file.

To complete the model-to-text transformation, MOCA serialises the AST by traversing it depth first:

- Folder and file nodes are created in the file system.

[7] Left → right, hence "forward".

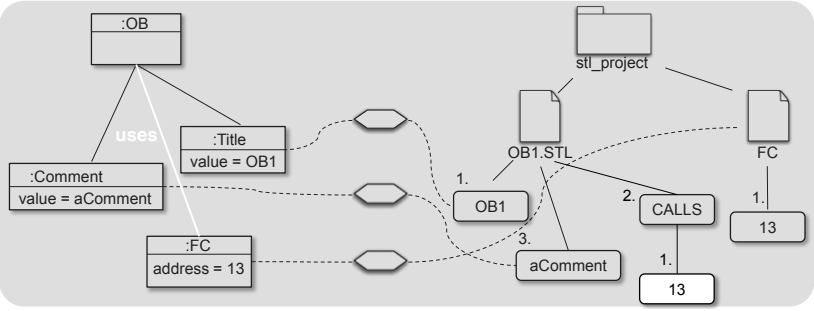

Figure 7: Model-to-text transformation after application of rule `transformCALLS`

- For every other node, a search is made in the template archive for a corresponding template for the node. If a template with the right name is found, it is instantiated and all child nodes are passed as template parameters.
- If no appropriate template was found, MOCA simply uses the value of the node as the default textual representation.

To complete the round-trip (Fig. 1) the user may now edit the generated code. For the text-to-model transformation, this modified code is passed to the parser to produce the corresponding AST (Fig. 3). With this AST now as the input model, the *backward graph tranformator* of the specified pair grammar (Fig. 4) would create the new STL inspired model analogously to the model-to-text transformation.

With this declarative and relatively simple grammar, our case study can be completed using TGGs as a model-to-model transformation language to integrate our STL inspired metamodel (Fig. 2) with the LOS metamodel and achieve our goal of integrating LOS models with corresponding STL code artifacts (Fig. 1). Furthermore, a central goal is to achieve – from a user perspective – a homogeneous and conceptually seamless change from model-to-text to model-to-model transformations as required for the complete tool integration (LOS $\leftrightarrow$ STL).

5 Summary and Outlook

In this paper, we identified a series of important open problems that we introduced with a real-world case study. As there is currently little or no support from existing approaches to address these challenges, we presented an informal description of a new bidirectional model-to-text transformation language, inspired by Triple Graph Grammars [Sch94] and the original concept of a Pair Grammar [Pra71], and argued benefits using our case study as a concrete example.

In future work, we will present a formal specification of our bidirectional model-to-text transformation language and underline similarities and differences to existing TGG theory. We will continue work on our prototype and strive to share a common core with our existing TGG implementation in MOFLON. As the current pattern matching engine [GSR05] in use relies heavily on types, we will implement pattern matching on the AST (generic and poorly typed) using different algorithms that do not require types and instead exploit the tree structure.

Numerous applications are planned including a MATLAB tool adapter [ALSS08], our current research cooperation with Siemens AG (LOS ↔ STL), our internal code generator in MOFLON to further bootstrap MOFLON with our own technology, and textual syntaxes for our modelling and transformation languages (MOF, TGG, SDM) that currently only have a graphical concrete syntax. With the experience obtained from all of these diverse use cases, we hope to be able to distill a set of common requirements and features (scopes, declaration and definition, primitive types, etc.) that can be supported directly as special *edges* and *nodes* in the AST fragments of our pair grammars. Last but not least, we regard a suitable, intuitive, graphical *and* textual syntax for our transformation language as very critical for its acceptance and success.

Bibliography

[AKK+08] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, A. Schürr. Metamodel-based Tool Integration with MOFLON. In *Proc. of the 30th ICSE*. Pp. 807–810. ACM Press, May 2008. Formal Research Demonstration.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Rensink and Warmer (eds.), *Proc. of the 2nd ECMDA-FA*. LNCS 4066, pp. 361–375. Springer, 2006.

[ALSS08] C. Amelunxen, E. Legros, A. Schürr, I. Stürmer. Checking and Enforcement of Modeling Guidelines with Graph Transformations. In *AGTIVE 2008*. LNCS 5088, pp. 313–328. Springer, 2008.

[BGSZ10] M. Bork, L. Geiger, C. Schneider, A. Zündorf. Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In *Proc. of the 4th ECMDA-FA*. LNCS 5095, pp. 33–47. Springer, 2010.

[EGL+05] *Model transformation by graph transformation: A comparative study*. 2005.

[EK06] S. Efftinge, C. Kadura. oAW 4.1 Xpand. Language reference. August 2006. http://www.eclipse.org/gmt/oaw/doc/

[EV06] *oAW xText: A framework for textual DSLs*. 2006.

[GBU09] T. Goldschmidt, S. Becker, A. Uhl. Textual Views in Model Driven Engineering. In *Proc. of the 35th SEAA*. Pp. 133–140. IEEE Computer Society, 2009.

[GBU10] T. Goldschmidt, S. Becker, A. Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Model Driven Architecture – Foundations and Applications*. LNCS 5095, pp. 169–184. Springer, 2010.

[GSR05] *Template-and modelbased code generation for MDA-Tools*. 2005.

[KLKS10] F. Klar, M. Lauder, A. Königs, A. Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. Springer, 2010. Accepted for publication.

[KRS09] F. Klar, S. Rose, A. Schürr. TiE - A Tool Integration Environment. In *Proc. of the 5th ECMDA Traceability Workshop*. CTIT Workshop Proceedings WP09-09, pp. 39–48. 2009.

[KRV08] H. Krahn, B. Rumpe, S. Völkel. Monticore: Modular development of textual domain specific languages. *Objects, Components, Models and Patterns*, 2008.

[LSRS10] M. Lauder, M. Schlereth, S. Rose, A. Schürr. Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. *Bulletin of the Polish Academy of Sciences, Technical Sciences*, 2010. Accepted for publication.

[Nag96] M. Nagl (ed.). *Building tightly integrated software development environments: the IPSEN approach*. Springer, 1996.

[Par04] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proc. of WWW '04*. Pp. 224–233. ACM, 2004.

[Par07] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[Pra71] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.* 5(6):560–595, 1971.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Tinhofer (ed.), *20th International Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 903, pp. 151–163. Springer, 1994.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In *Proc. of the 4th ICGT*. LNCS 5214, pp. 411–425. Springer, Nov. 2008.

[SRS09] M. Schlereth, S. Rose, A. Schürr. Model Driven Automation Engineering - Characteristics and Challenges. In *5th Workshop on MBEES*. 2009.

[SV05] T. Stahl, M. Voelter. *Model-Driven Software Development*. Wiley, 2005.

[Wag09] R. Wagner. *Inkrementelle Modellsynchronisation: Univ., Diss.–Paderborn, 2009.* Logos-Verlag, 2009.