A Framework for Bidirectional Model-to-Platform Transformations

Anthony Anjorin*, Karsten Saller, Sebastian Rose, and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{anjorin,saller,rose,schuerr}@es.tu-darmstadt.de

Abstract. Model-Driven Engineering (MDE) has established itself as a viable means of coping with the increasing complexity of software systems. *Model-to-platform* transformations support the required abstraction process that is crucial for a model-driven approach and are, therefore, a central component in any MDE solution. Although there exist numerous strategies and mature tools for certain isolated subtasks or specific applications, a general *framework* for designing and structuring model-to-platform transformations, which consolidates different technologies in a flexible manner, is still missing, especially when *bidirectionality* is a requirement.

In this paper, we present: (1) An abstract, conceptual framework for designing and structuring bidirectional model-to-platform transformations, (2) a concrete instantiation of this framework using string grammars, tree grammars, and triple graph grammars, (3) a discussion of our framework based on a set of core requirements, and (4) a classification and detailed survey of alternative approaches.

Keywords: bidirectional model-to-platform transformations, string grammars, tree grammars, triple graph grammars.

1 Introduction

Model-Driven Engineering (MDE) has established itself as a viable means of coping with the increasing complexity of modern software systems by increasing productivity, supporting platform independence and interoperability, and reducing the gap between problem and solution domains [2].

Model transformations, in general, play a central role in any model-driven solution [2] and *model-to-platform* transformations, in particular, enable an abstraction from platform-specific details, which is usually an important first step in the Model-Driven Architecture (MDA) approach [2]. In this paper, a *platform* is defined as the final step in a given transformation chain and, in this context, includes textual files (XML files, configuration files, property files and code in a

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

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 124–143, 2013.

programming language), folder and file hierarchies (structures in a filesystem), engineering tools with internal data structures that can be manipulated via an API, and very simple and typically generic tree-like structures. A *model* is an abstraction that is suitable for a particular purpose. We regard models as being conform to a *metamodel*, which is a representation of relevant concepts and relations in a domain usually specified with a standard modelling language such as UML[1], MOF[2] or Ecore [20].
Application areas of model-to-platform transformations include:

1. Round trip engineering involving code generation (forward engineering), and system comprehension (reverse engineering).
2. The development and evolution of Domain Specific Languages (DSLs).
3. The integration of different tools with tool-specific import/export formats.

As these applications are and always have been crucial tasks in software engineering, various approaches and tools already exist [8,9,19]. Current approaches are, however, either application specific (e.g., fixed metamodel), only handle an isolated subtask (e.g., only code generation), or are strongly tied to a certain technology or standard (e.g., Ecore). A general *framework* that can be used to design and structure model-to-platform transformations in a flexible manner is, therefore, still missing, especially when *bidirectionality*, crucial in many applications [5,12], is an important requirement. Such a framework must combine and consolidate state-of-the-art technologies in such a way that the strengths of individual components are emphasized and weaknesses are compensated, but still be general enough to allow a free choice and replacement of concrete standards or technologies. In this paper, we present:

1. An abstract, conceptual framework for structuring the required components of a bidirectional model-to-platform transformation.
2. A concrete instantiation of this framework based on string grammars and tree grammars using ANTLR [18], and Triple Graph Grammars (TGGs) [14] using eMoflon [1] and the Eclipse Modeling Framework (EMF) [20].
3. A discussion of our framework based on a set of core requirements.
4. A classification and detailed survey of alternative approaches.

The paper is structured as follows: Section 2 presents our running example, discusses further application domains for bidirectional model-to-platform transformations, and identifies a set of core requirements. Section 3 introduces an abstract conceptual framework, independent of any concrete technologies, together with a concrete instantiation thereof as a proof-of-concept. Section 4 classifies related approaches and compares them with our framework based on our requirements, while Sect. 5 concludes the paper with a summary and a brief overview of future work.

[1] Unified Modeling Language.
[2] Meta-Object Facility.

2 Application Domains and Core Requirements

Model-to-platform transformations are relevant in a multitude of application domains. In this section, we focus on application areas that additionally involve bidirectionality and derive a minimal set of *core* requirements.

Our running example is taken from the application domain *Round Trip Engineering* and is used consequently in the rest of the paper to introduce and explain all relevant concepts.

2.1 Running Example: Round Trip Engineering

Inspired by a real-world system modernization and re-engineering application scenario[3], our running example involves a software developer (Fig. 1::1[4]) who is trying to improve a software system that consists of a substantial number of source code files (Fig. 1::2). The developer has a set of *refactoring rules* (Fig. 1::3) which are to be automatically applied to the software system to result in an improved version (Fig. 1::4). For our concrete example, each source code file specifies a number of *components*, each of which can *require* other components.

To support system comprehension, e.g., in preparation of a re-engineering of the system, a *dependency analysis* of the complete system (possibly comprising thousands of components) is required. A suitable metamodel that captures the relevant concepts needed to express the dependencies in the system (Fig. 1::5) is established and a *platform-to-model* transformation (Fig. 1::6) is used to extract a dependency graph (Fig. 1::7) from the software system. The refactoring rules can now be expressed as a model transformation (Fig. 1::8) that can be applied to yield an improved dependency graph (Fig. 1::9). The final step is to update/regenerate the system with a *model-to-platform* transformation (Fig. 1::10).

To keep things simple, we restrict the analysis to only the component dependency graph and ignore the internal specification of each component. Although components can require components in different source files, we restrict the analysis to a single file for presentation purposes. The sample file depicted in Fig. 2 consists of four components `T`, `L`, `R` and `B`. The components form a dependency *diamond* as `B` requires `T` indirectly via `L` and `R`. Due to certain domain specific reasons (e.g, redundant memory allocation for the topmost component), our client wishes to avoid such diamond dependency subgraphs. The refactoring rule for our running example is as follows: If the component at the base of the diamond (`B`) is not required by any other components, it should be copied[5] and one of the dependencies (`R`) must be transferred to the copy (`B_Copy`) to break up the diamond. The refactored file according to this rule is depicted in Fig. 3.

A model-driven approach is advantageous as the metamodel is an abstraction, which can be chosen to be exactly suitable for the task, i.e., the refactoring

[3] Part of an industrial cooperation with Eckelmann AG (`www.eckelmann.de`)

[4] The notation Fig. n::m refers to label m in Fig. n.

[5] A further simplification as, in reality, an analysis of the *content* of the component is required to determine how it can be appropriately split into two independent parts.

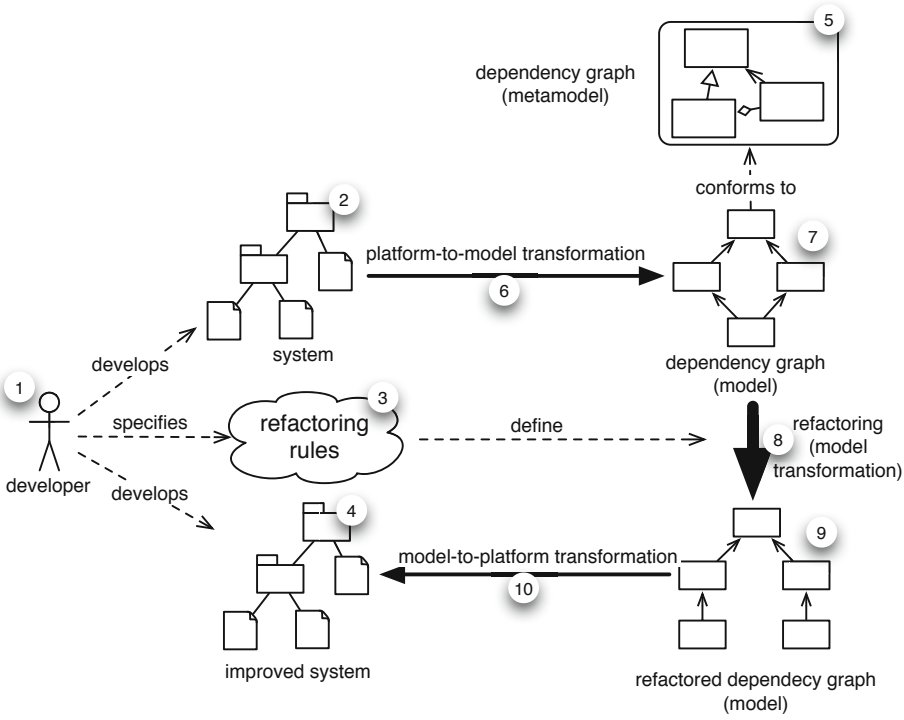


Fig. 1. Overview of the running example

```
1  Component T {
2    // ...
3  }
4  Component L requires T {
5    // ...
6  }
7  Component R requires T {
8    // ...
9  }
10 Component B requires L R {
11   // ...
12 }
```

Fig. 2. Before refactoring

```
1  Component T {
2    // ...
3  }
4  Component L requires T {
5    // ...
6  }
7  Component R requires T {
8    // ...
9  }
10 Component B requires L {
11   // ...
12 }
13 Component B_Copy requires R {
14   // ...
15 }
```

Fig. 3. After refactoring

rules can be expressed as model transformations in a very concise, readable and maintainable manner. Furthermore, *bidirectionality* is a crucial requirement as the results of the model transformations must be reflected in the source code.

Our running example from the domain of round-trip engineering shows that bidirectional model-to-platform transformations are crucial to support the automation of repetitive, boring tasks (manually refactoring the source code files). In the following, we give a brief overview of other application domains which we use to derive a set of core requirements.

2.2 Further Application Domains

Round-trip engineering is not the only application domain where bidirectional model-to-platform transformations are necessary. From numerous examples we choose two further domains and give a schematic representation of the workflow and requirements for bidirectionality in each case.

DSL Development and Evolution: Domain Specific Languages (DSLs) are programming languages of limited expressiveness, which are designed to be maximally suitable for a particular task or domain [8]. The systematic development of such languages to increase productivity and improve communication between domain experts and professional software developers is a major application of MDE technologies in general, and model transformations in particular.

Models that conform to a certain metamodel (Fig. 4::1) can be specified in *abstract syntax*, i.e., as typed attributed graphs with respect to the metamodel (the type graph). For domain experts who wish to create models in the DSL, a *textual concrete syntax* is a more suitable means of specifying models, and supporting this requires at least a unidirectional text-to-model transformation (Fig. 4::2).

There are two reasons why bidirectionality is an important requirement in this context: Firstly, there might be a different group of domain experts who prefer to use some other kind of concrete syntax (possibly visual) to specify models of the same DSL (Fig. 4::3). To exchange models freely between the different groups of experts, it must be possible to transform back and forth, which requires a bidirectional model-to-platform transformation for each supported concrete syntax of the DSL. Secondly, a DSL *will* evolve over time to accommodate new or changed requirements and this can be supported via a corresponding model transformation from the old version of the metamodel to a new one (Fig. 4::4). In a real-world scenario, all models specified in the old version of the DSL (Fig. 4::2), *must* be transformed to be conform to the new metamodel and a possibly new version of the textual concrete syntax (Fig. 4::5). Such DSL evolution support clearly requires a bidirectional model-to-platform transformation for each version of the DSL.

Tool Integration: Engineering processes typically involve different stakeholders who work together to build or maintain a system. Each stakeholder has a specialized viewpoint or specific needs regarding the complete engineering process and uses established engineering tools in the corresponding (sub)domain.

An efficient exchange of data and information between the tools in use, i.e., *tool integration* [21], avoids redundancy and ensures consistency across tool borders.

A schematic tool integration setup is depicted in Fig. 5. An engineer in a certain domain works with his preferred engineering tool A (Fig. 5::1). To exchange data with another engineer using a different tool B (Fig. 5::7), the relevant data for the integration must be extracted from the tools. Engineering tools are very often *commercial off-the-shelf* tools and might only offer a tool-specific import/-export format (Fig. 5::2), very often XML [15]. In a tool integration scenario, a *tool adapter* (Fig. 5::3) is required to extract a suitable model (Fig. 5::4) from the available textual exchange format via a bidirectional model-to-platform transformation. The extracted model can then be synchronized (Fig. 5::5) with different but related models from other tools (Fig. 5::6), updating the data in the tools via respective tool adapters. Bidirectionality is thus an important requirement.

2.3 Core Requirements

After highlighting important application domains for bidirectional model-to-platform transformations, we derive a core set of requirements in this section, divided into two main groups: (1) Requirements concerning the resulting bidirectional model-to-platform transformation that is to be implemented, and (2) requirements concerning the *process* of establishing such a transformation.

Requirements Concerning the Resulting System: All requirements are formulated for a "correct" system, i.e., we assume that the transformation must perform as specified, e.g., by a testsuite, before requirements are considered.

(R1) Maintainability: The most important requirement is that the complete transformation be maintainable. This means that it should be relatively easy to extend, improve or otherwise adapt the transformation for all participants and stakeholders. This implies a number of sub-requirements including:

- *Support for bidirectionality* to ensure that changes in requirements can be reflected in the system without introducing inconsistencies,
- *Readability* to support communication and knowledge transfer amongst developers and to allow for a validation by domain experts who might not be professional software engineers and must at least understand parts of the transformation,
- *Stability* allowing subsystems to be exchanged as required without causing ripple-effects in others.

(R2) Scalability: Depending on the exact limits posed by the application domain and concrete scenario, the transformation must scale with respect to memory consumption and runtime complexity. If possible, this should be *guaranteed* by the applied approach, e.g., polynomial runtime.

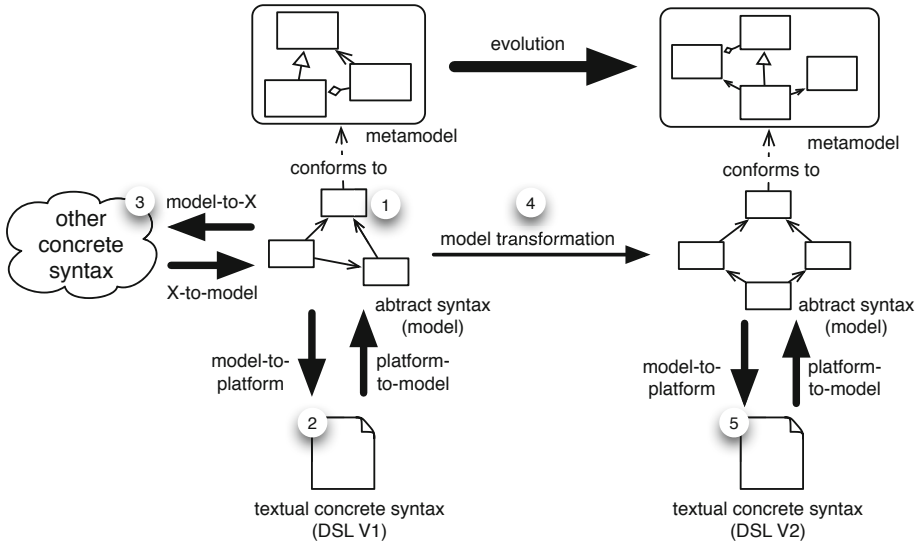


Fig. 4. DSL Development and Evolution

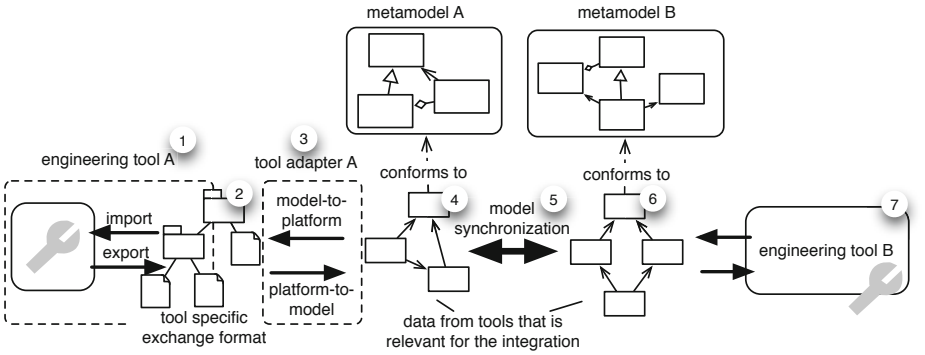


Fig. 5. Tool Integration

Requirements Concerning the Process: For the actual process involved in establishing a bidirectional model-to-platform transformation, we consider the following two requirements.

(R3) Productivity: The most important requirement concerning the development process is productivity, i.e., the speed of development which implies *usability*, adequate *tool support*, the possibility to *iteratively* develop and improve the system, and support for testing (static analysis for validation, debugging).

(R4) Generality: To handle real-world applications, the approach must be *general* enough. This means that the restrictions posed by the approach should not be too

limiting, i.e., should not restrict the class of possible applications to such an extent that the approach becomes useless for most practical purposes. This requirement has two implications: The approach must offer (i) a flexible and well-defined *fallback* to a turing complete language for situations where restrictions cannot be met, and (ii) a *uniform* treatment of a wide range of platforms including XML and other textual formats, directory structures, and generic data structures.

3 The Moflon Code Adapter (MOCA) Framework

Figure 6 depicts a framework for organizing the components necessary for a bidirectional model-to-platform transformation. This framework is abstract in the sense that it does not prescribe any concrete technologies or modelling standards. The main idea of our approach is to separate the transformation into two distinct parts: (i) A platform-to-tree transformation and (ii) a tree-to-model transformation.

The *platform* (Fig. 6::1) is transformed via a *parser* (Fig. 6::2) to a simple tree structure (Fig. 6::3). This tree structure should be a minimal abstraction of the platform which is nonetheless accessible to the chosen bidirectional transformation language (Fig. 6::4). Trees are transformed back to the platform via an *unparser* (Fig. 6::2) which typically linearizes the tree structure and adds platform details which were abstracted away by the parser if necessary. A crucial point is to keep the parser and unparser *as simple as possible*. In our opinion, this first step is often not worth supporting with a bidirectional language[6], and,

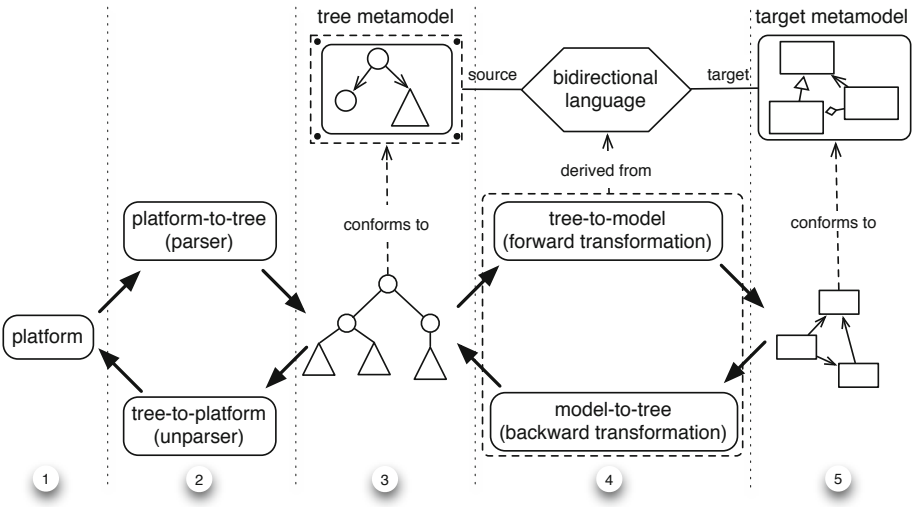


Fig. 6. The Moflon Code Adapter (MOCA) Framework

[6] This is not always true, e.g., if the application scenario requires layout preservation.

if it is kept to a bare minimum, almost all complexity can be shifted to the model-to-tree transformation, which can be appropriately handled with a suitable bidirectional transformation language (Fig. 6::4). In an MDE context, the tree should be a very simple structure which is nonetheless already conform to the modelling standard as required by the bidirectional transformation language and the target model (Fig. 6::5). As almost all standard parsers are context-free, "very simple" usually means acyclic and homogeneous with respect to typing (i.e., very few or even only a single "node" type is used).

The process prescribed by the framework already has a number of advantages:

Separation of Concerns: A strict separation of platform *comprehension* and *generation* (Fig. 6::2) from the actual *transformation* (Fig. 6::4) positively affects maintainability (R1) and productivity (R3) as the (un)parser can be kept very simple and be replaced without having to change the transformation. Furthermore, the bidirectional language can operate on a tree structure without irrelevant details of the textual representation leading to a simpler transformation with the clear task of (i) adding appropriate typing information and (ii) deducing context-sensitive relations to obtain the target model (Fig. 6::5).

A Clear Interface to Different (un)parser Technologies: Establishing a simple tree structure for the bidirectional transformation consolidates XML and different abstract syntax trees produced by parsers. Even the directory and file structure can be embedded in the tree structure if it is relevant for the transformation. This positively affects the generality (R4) of the approach.

Demanding only a semi-structured, i.e, hierarchical structure, greatly simplifies the task of parsing and unparsing and clearly places most of the complexity in the transformation, which can be supported with a bidirectional language. This applies the right tool for the right job and also allows for using standard parser and unparser technology via simple adapters, which can be easily replaced. This positively affects maintainability (R1) and productivity (R3).

Modularity: In general, the modular structure of the framework enables a high level of reuse and exchangeability of the platform, parser and/or unparser, the model-to-tree transformation, the target metamodel and the modelling standard without affecting all other components. This positively affects maintainability (R1) as components are stable, productivity (R3) due to possible reuse, e.g., of existing (un)parsers, and generality (R4), as at least parts of the system can be ported to a different platform or standard.

3.1 An Implementation of MOCA in eMoflon

As a proof-of-concept, the MOCA framework has been realized as part of our metamodelling tool eMoflon [1] and can be downloaded and used as described in our detailed tutorial[7]. Our MOCA implementation is currently in use for

[7] Available from `www.moflon.org`

a lecture at our university, and in two ongoing projects handling real world applications[8] from the industry.

Figure 7 depicts the concrete instantiation of the abstract MOCA framework as realized in eMoflon. The supported platform (Fig. 7::1) is a directory structure, which can contain files with different textual content as indicated by the shading in the diagram. To complement a built-in directory "parser", the user must provide a parser (Fig. 6::2) for each type of file, to produce an abstract syntax tree which is inserted as a shaded subtree into the resulting tree (Fig. 6::3). Our MOCA implementation provides dedicated support for XML via an adapter layer, i.e., arbitrary XML files can be automatically transformed to instances of our tree metamodel (`MocaTree`). To handle arbitrary textual formats, we provide support for parsers and unparsers generated with ANTLR [18] via *string grammars* and *tree grammars* with templates [19], respectively.

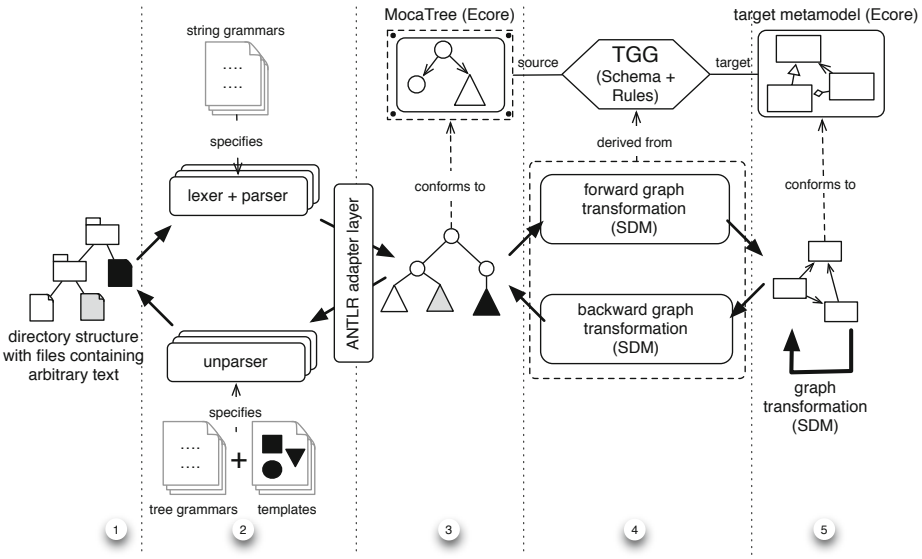


Fig. 7. A realization of the MOCA Framework as part of eMoflon

The textual format for our running example (Fig. 2 and 3) is non-standard and requires a parser. Using ANTLR, this involves specifying a lexer and parser as depicted for the example in Fig. 8. Please note how the parser builds up an abstract syntax tree indicated in the textual syntax as `^(ROOT CHILDREN)`. Using our ANTLR MOCA adapter, the abstract syntax tree produced by the parser can be directly loaded as an EMF model without any further effort. Please note that the lexer and parser *do absolutely nothing else* apart from recognizing the textual content and building a simple, homogeneous hierarchical structure.

[8] A bidirectional RTF to HTML transformation and a common DSL for consolidating iOS and Android app development.

```
COMPONENT:  'Component';
SPEC:       'SPEC';
BODY :      '{' .* '}'
REQUIRES:   'requires';
ID:         ('a'..'z'| 'A'..'Z')+;
WHITESPACE: ('\t' | ' ' | '\r' | '\n')+
```

(a) Lexer Grammar

```
main: componentSpec+
  -> ^(SPEC componentSpec+);
componentSpec: COMPONENT ID dep? BODY
  -> ^(ID ^(REQUIRES dep?) BODY);
dep: REQUIRES reqs+=ID+
  -> $reqs+;
```

(b) Parser Grammar

Fig. 8. Lexer and Parser Grammars

For code generation, the context-free nature of the tree is exploited using a *tree grammar*, which traverses the structure of the tree in a depth-first manner and evaluates a set of templates to produce text (Fig. 9). We use *StringTemplate* [19] as a template language, which is a very restricted, extremely simple template language with a minimal set of commands. Enforcing such simple templates leads to a strict model-view separation with various advantages [17]. These four simple rule-based specifications (Fig. 8, Fig. 9) implement the first step in the framework, the platform-to-tree transformation (Fig. 7::2), for our example.

```
main:  ^('SPEC'
         content+=component*)
  -> file(content={$content});

component: ^(name=STRING
             r+=reqs
             body=STRING)
  -> component(name={$name},
               r={$r},
               body={$body});

reqs: ^('REQUIRES'
        l+=STRING*)
  -> reqs(l={$l});
```

(a) Tree Grammar

```
file(content) ::= <<
<content; separator="\n\n">
>>

component(name, r, body) ::= <<
Component <name> <r>{<body>}
>>

reqs(l) ::= <<
<if(l)>
requires <l; separator=" ">
<endif>
>>
```

(b) Templates

Fig. 9. Tree grammar and templates

In our MOCA implementation, we use *Triple Graph Grammars* (TGGs)[14] as the bidirectional language to transform the context-free, homogeneous "tree" (Fig. 7::3) to the target model (Fig. 7::5). eMoflon is EMF/Ecore based and thus, the modelling standard used is EMF. Figure 10 depicts the *TGG Schema* for the running example, which is a triple of the metamodels involved in the transformation. To the left, the tree consists of files and nodes, while our target model, to the right, consists of "components" which can require other components and are all contained in a specification. Our complete tree metamodel

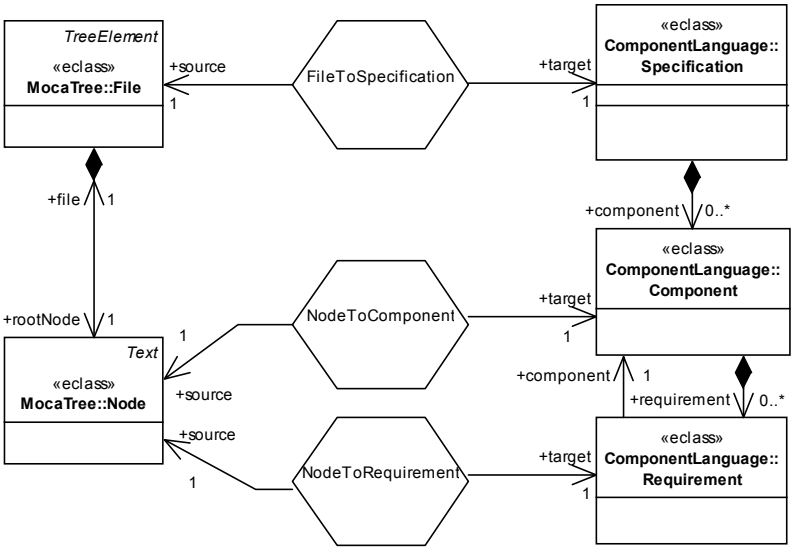


Fig. 10. TGG Schema (Triple of involved metamodels)

is just complex enough to represent XML files, directory structures and parse trees without losing information, i.e., we have basic concepts of folders, files, and nodes with attributes and labels. In the middle, *correspondence* types are defined, which are used for traceability. Already from the schema, it is clear that each file corresponds to a specification, and that both components and requirements correspond to nodes in the tree.

A TGG specification consists of a schema and a set of rules that describe the simultaneous evolution of triples of connected source, correspondence and target models. The advantage of using TGGs is that both forward and backward transformations can be automatically derived from this single specification and are guaranteed to be compatible with the described simultaneous evolution.

In sum, the required transformation for our running example consists of three TGG rules, one to transform files and specifications, one to handle components and one to create the requirement relation between components. The latter is depicted in Fig. 11. Please note that the hexagonal shape of correspondence types in the TGG schema and correspondences in the TGG rule is just syntactic sugar to indicate at a glance that these objects belong to the correspondence domain, i.e., can be interpreted as traceability types and links, respectively.

A TGG rule consists of context elements, depicted in black without any stereotype, and create elements depicted in green with an additional `create` stereotype. Context elements must be created by other rules and are used to induce an implicit dependency between rules, e.g., the TGG rule `NodeToRequirement` can only be applied if the two components involved (`component` and `reqComponent`) have already been created and identified with nodes in the tree by other rules (Fig. 11). The rule creates a requirement between `component` and

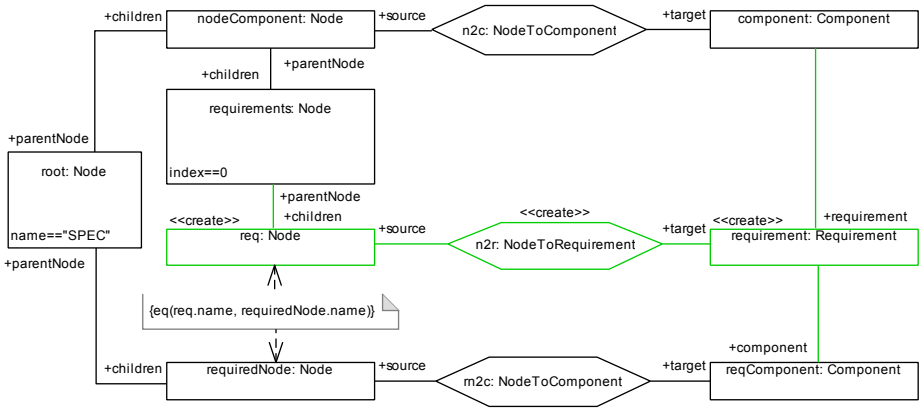


Fig. 11. TGG Rule NodeToRequirement

`reqComponent`, reflecting this in the tree by creating a new requirements node `req` for `nodeComponent`, with its name equal to that of `requiredNode`. This condition is expressed using the *attribute constraint* `eq(req.name, requiredNode.name)`. The TGG rule `NodeToRequirement` showcases the two main tasks of the bidirectional transformation: (i) Introducing appropriate typing, e.g., a `Requirement` instead of just a `Node`, and (ii) replacing the context-free acyclic tree with a context-sensitive graph structure, e.g., connecting two components via a requirement directly instead of having separate nodes with the same name.

To complete our running example, we can now specify the refactoring rules using an appropriate transformation language that can operate on the target model. Figure 12 depicts a graph transformation rule using *Story Driven Modelling* (SDM)[7], our graph transformation language for unidirectional model transformations in eMoflon. The rule is declarative and concise, and the diamond structure to be found can actually be "seen". According to the refactoring rule (Fig. 12), the dependency diamond should only be resolved if `component` is not required by any other components. This is enforced using a *negative application condition* (NAC) depicted by the crossed out element `otherReq`.

If the diamond structure is found and the NAC is not violated, the requirement `req2` is relocated to a new component `newComponent`, created as a copy of `component`. Please note the stereotype `destroy/create` used to indicate elements that should be deleted/created[9], the *attribute assignments* used to initialize the attribute values in `newComponent`, and the fixed (bound) elements starting from which the other elements must be found, indicated with a thicker border (`component` and `this`).

After applying the rule to all components, the refactored model can be transformed back to a tree and used to generate the refactored textual file as required (Fig. 3). After a backward or forward transformation, the created triple

[9] Additionally indicated via the red/green colour of the elements.

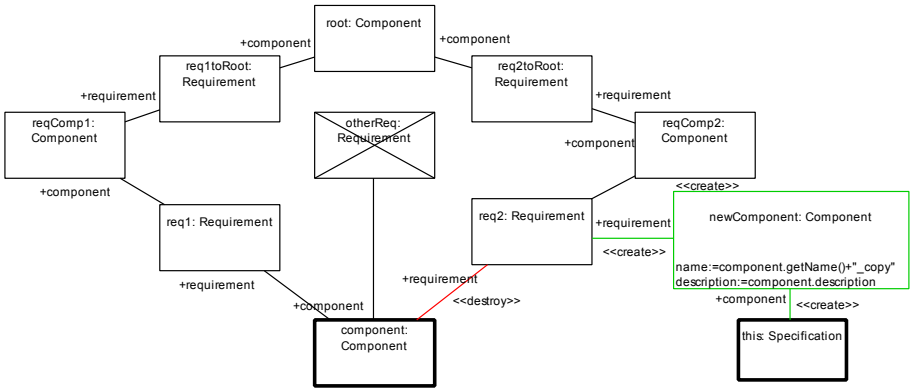


Fig. 12. SDM refactoring rule

of source, correspondence and target models can be visualized in eMoflon using our *integrator* which represents the correspondence model visually as links.[10]

In addition to the advantages of the abstract MOCA framework, our concrete choice of languages and standards has the following advantages:

Homogeneity via Complementary Languages: Our choice of languages, i.e., string grammars, tree grammars combined with templates with minimal logic, triple graph grammars and SDMs are all *rule-based* and *declarative*[11]. Based on our experience of working with this mix of languages, we believe that a high level of homogeneity is attained by supporting a common rule-based thinking in *patterns*. This positively affects maintainability (R1) and productivity (R3) as there is no disturbing shift in paradigm and trained skills in one language can be transferred to all others.

Formal Properties and Guarantees: By separating the transformation into different steps, the formal properties of the different individual languages still hold for the corresponding step. For example, the runtime complexity for *LL(*)* parsing (worstcase $O(n^2)$, in practice actually much less [18]) holds for tree generation, while TGGs guarantee polynomial runtime [14] for the tree-to-model transformation. Depending on the application scenario, such runtime efficiency can be crucial for scalability (R2). Furthermore, TGGs also guarantee that the derived transformations are *correct* with respect to the specified TGG, i.e., only a single specification is used, which positively affects maintainability (R1).

Flexible Fallback to Java: In our experience, practical problems can almost never be *completely* solved with a DSL. For example, even if a large part of a transformation can be specified with TGGs, certain parts, especially low-level

[10] Please refer to our tutorial (`www.moflon.org`) for screenshots and further details.

[11] SDMs are *programmed* graph transformations and, therefore, also have usual imperative language constructs such as if-else and loops.

attribute manipulation, must be specified using SDMs or directly in Java. All languages used in our MOCA implementation support a *fallback* to a more general language when necessary: ANTLR offers *syntactic* and *semantic predicates* in string and tree grammars [18], which can be used to embed Java statements to support the lexer/parser, SDMs offer *MethodCallExpressions* [1] to mix Java code in graph transformations in a type safe manner, and TGGs offer *Attribute-Constraints*, which are implemented in Java. ANTLR and eMoflon are both completely generative, i.e, map specifications to standard Java code which also simplifies mixing in hand-written code. This positively affects generality (R4), as basically any problem can be tackled that could also have been solved directly in a general purpose language, in our case Java.

Iterative Workflow: Last but not least, an iterative workflow is possible as the target metamodel can be iteratively refined. In each step, more parts of the tree can be handled by TGG rules until the transformation is complete. The platform can also be handled in an iterative manner, e.g., by using regular expressions to "filter" the textual files in the first iterations instead of a parser. ANTLR also supports this with a "fuzzy" parsing mode that ignores all content that cannot be parsed, i.e., parses these parts as a string block without further processing/structuring. An iterative workflow improves productivity (R3) as most mistakes can be found early enough in the development process.

3.2 Limitations and Drawbacks

Every approach has limitations and in the following, we discuss the most important drawbacks of our framework and its concrete implementation in eMoflon:

A Steep Learning Curve: Separating the transformation into different steps that are implemented with different languages has the potential of increasing complexity in general. Although we have tried to choose complementary languages with a common paradigm, it is still challenging to master all the different languages, especially without prior experience with rule-based languages.

Requires a Model-to-Model Transformation: Requiring a model-to-tree transformation as a separate step introduces an extra transformation language (TGGs) and tool dependency in the transformation chain.

Incrementality: The separation in different parts advocated by our framework makes the task of supporting incrementality for the *complete* transformation chain more challenging than if all steps were merged in a single specification.

4 Related Work

In this section, we discuss the main groups of alternative approaches to our generalized tree-based approach and highlight main strengths and weaknesses. We do not try to give a complete list of concrete tools but rather focus on *groups* of approaches, mentioning a few concrete representatives in each case.

4.1 Combination of Unidirectional Approaches

Although there is an increasing number of bidirectional languages available, the standard way of implementing bidirectional model-to-platform transformations is still to use two unidirectional transformation languages, one for each direction. Typical combinations include *Xtext* [6] for platform-to-model and *Xpand*[12] for model-to-platform, or *ANTLR* for platform-to-model and *Velocity*[13] for model-to-platform. The main advantage is clear; A combination of standard, mature unidirectional approaches is very general (R4) and "gets the job done" while existing bidirectional approaches are mostly still in development and are often not usable for real-world application scenarios, although they might work very well for a restrained class of problems. The flexible combination and the possibility of implementing parts of the transformation in standard languages offered by our framework is, in this respect, a pragmatic approach to having the best of both worlds, i.e., still profiting from the advantages of a bidirectional language. Similarly, standard approaches typically scale well (R2) with respect to runtime and memory consumption. A challenge, however, is handling *incremental* changes which becomes difficult when separate tools are used for each direction. Scalability then becomes a major problem if the scenario involves a *synchronization* in contrast to a *batch transformation*.

A further disadvantage of a combination of unidirectional approaches is that it is hard to maintain (R1): Changes to the forward transformation have to be carefully reflected in the backward transformation and vice-versa, and this gets increasingly difficult with the complexity of the transformation. Productivity (R3) also suffers as two separate specifications have to be implemented. A bidirectional language would be advantageous in both cases.

4.2 Tightly Integrated Software Development Environments

A second group of approaches are tightly integrated software development environments that provide *view-based*, *syntax directed* editing, keeping the concrete and abstract syntax of models synchronized at all times. This means that the editor operates directly on the abstract syntax of a model and reflects changes immediately in the presented concrete syntax (the view). Examples for such environments include *Furcas* [10], *MPS* and *Ipsen* [16].

A syntax directed editing approach usually has rich support from the corresponding framework/environment with which the transformation can easily be specified, i.e., although this depends on the concrete environment, the process is usually quite productive (R3) and the resulting transformation is maintainable (R1) as it is truly bidirectional. Scalability (R2), especially with respect to memory consumption, again depends on the concrete environment, but *incrementality* can easily be supported with such a tightly integrated approach.

A disadvantage is that there is a high dependency on the enclosing framework. This becomes problematic when the transformation is to be ported to a new

[12] OpenArchitectureWare, `http://www.eclipse.org/gmt/oaw/`
[13] The Apache Jakarta Project, `http://jakarta.apache.org/velocity/`

modelling standard or a component has to be replaced. A further disadvantage is that an on-the-fly synchronization of concrete and abstract syntax might not be possible in some application scenarios, as text files might have to be changed "offline". Furthermore, most approaches in this group are geared towards DSL development and are not suitable for, e.g., a scenario where large parts of static text must be generated, which is more suited for template-based code generation.

4.3 Grammar-Based Approaches

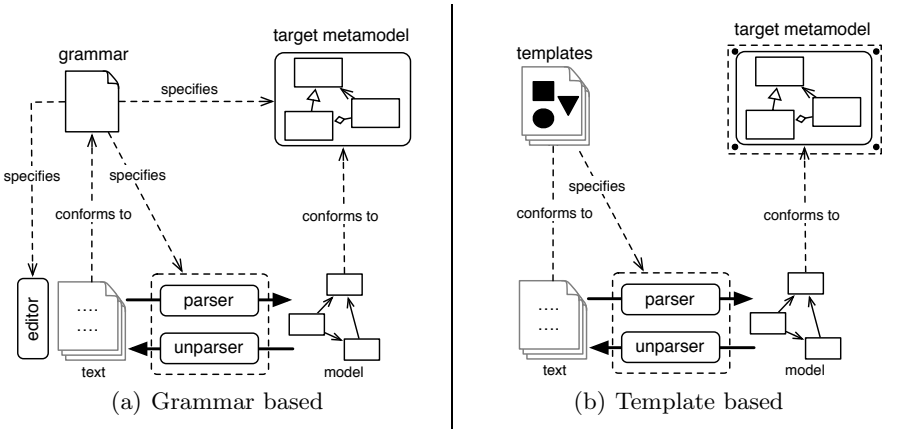
Grammar-based approaches such as *Xtext*, *Spoofax* [13], and *Monticore* [11] provide appropriate extensions to EBNF to allow context-sensitive relationships to a certain extent. As depicted in Fig. 13(a), the main idea is to derive as much as possible from the *grammar*, i.e., not only a parser, but also a metamodel, an editor, and an unparser. A metamodel can be extracted from the grammar either via an implicit transformation from EBNF to a modelling language (Ecore in the case of *Xtext*), or by extending EBNF to a complete modelling language which can be used to specify both the textual concrete syntax *and* the abstract syntax of the language combined in the grammar. The latter approach is taken by *Monticore*. Bidirectionality can be supported by using the non-terminals in the grammar to *pretty print* model elements to text.

Grammar-based approaches lead to very compact, concise specifications and are highly productive when the target language can be described with the grammar. Getting an editor "for free" is also a major productivity boost, especially when developing a textual DSL. In general, however, every grammar can only describe a limited class of languages, and, due to the fact that the grammar is used to derive all other components, a fallback to Java similar to what ANTLR offers cannot be supported. Every realistic transformation, therefore, will always require a subsequent model-to-model transformation, especially when the target metamodel was established *before* the textual syntax. In many cases, e.g., round-tripping as opposed to DSL development, the textual syntax *and* the target metamodel are fixed and already exist. In such a case it becomes quite challenging to specify a perfectly fitting grammar.

Supporting bidirectionality is also difficult in complex cases and most approaches do not place a strong focus on bidirectionality, only providing a default pretty printer that must be extended and refined. The price of having a compact, concise specification is that all components are merged making it difficult, if not impossible, to reuse the text comprehension part of the grammar for a different target metamodel, or to change the textual syntax but retain the same metamodel. Last but not least, grammar-based approaches are not suitable for cases where a lot of text is to be ignored or filtered and when a lot of static parts are to be generated.

4.4 Template-Based Approaches

Template-based approaches such as *Xround* [4] and [3] provide an interesting contrast to grammar-based approaches by deriving the complete bidirectional



(a) Grammar based (b) Template based

Fig. 13. Schematic overview of grammar-based and template-based approaches

transformation from a set of templates. As depicted in Fig. 13(b), a set of templates in a fixed template language is used to derive an unparser (code is simply generated with the templates) *and* a parser. The parser works by matching text fragments with potential templates until the exact sequence of chosen templates can be identified. Corresponding model elements can be derived from this sequence of templates as the target metamodel is fixed and known to the parser, i.e., there is a mapping between model elements and templates.

In contrast to a grammar-based approach, this works quite well for cases with large parts of static text which must be ignored/generated. For a typical textual DSL, however, with almost a 1-1 relationship between text and model elements for conciseness, the templates must contain a lot of logic and not so much static text, reducing readability and maintainability of the templates.

Although the generated textual syntax can be flexibly varied, the parser can only be realized efficiently if the template language *and* the target metamodel are fixed. This means that a template-based approach is a productive, maintainable solution for a *fixed metamodel*, i.e., a concrete application. The parser would, however, have to be almost completely re-implemented for every new metamodel. Depending on the complexity of the supported template language, it can also be quite challenging to parse textual content using templates in a scalable manner, i.e., complex logic in the templates can easily lead to an explosion of the template search space.

5 Conclusion and Future Work

In this paper, we presented a flexible, general framework for structuring bidirectional model-to-platform transformations. A set of core requirements derived from typical application domains was used to argue the advantages of a clear separation of the transformation into two distinct steps: A text-to-tree transformation (text

comprehension/generation) which is held as simple as possible, and a tree-to-model transformation (typing and context-sensitive relations), which should be implemented with a bidirectional language. A realization of our framework in eMoflon[14] shows that a flexible blend of rule-based, declarative languages can be combined successfully. Existing approaches for bidirectional model-to-platform transformation are either not general enough, i.e., only work for a certain standard/domain, or not flexible enough, i.e., components cannot be exchanged. Our choice of TGGs as a bidirectional language opens up a large class of applications for TGGs, with new challenges. Future tasks include improving support for incrementality in our TGG implementation by exploiting the asymmetric nature of model-to-platform transformations (information loss is only in one direction) as compared to the general case. We are also working on optimizing our current TGG algorithm to deal with weakly typed trees, necessary for an efficient inference of context-sensitive relations, and are investigating concepts for improving modularity and reuse/composition of the transformation languages (string grammars, tree grammars, templates, TGGs).

References

1. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: Leveraging EMF and Professional CASE Tools. In: Heiß, H.U., Pepper, P., Schlingloff, H., Schneider, J. (eds.) Informatik 2011. LNI, vol. 192, p. 281. GI, Bonn (2011)
2. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Feather, M., Goedicke, M. (eds.) ASE 2001, pp. 273–280. IEEE, New York (2001)
3. Bork, M., Geiger, L., Schneider, C., Zündorf, A.: Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 33–47. Springer, Heidelberg (2008)
4. Chivers, H., Paige, R.F.: XRound: Bidirectional Transformations and Unifications Via a Reversible Template Language. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 205–219. Springer, Heidelberg (2005)
5. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
6. Efftinge, S., Völter, M.: oAW xText: A Framework for Textual DSLs. In: EclipseCon Europe 2006 (2006)
7. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
8. Fowler, M.: Domain-Specific Languages. Addison-Wesley, Boston (2010)
9. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 169–184. Springer, Heidelberg (2008)

[14] Available from `www.emoflon.org`

10. Goldschmidt, T., Becker, S., Uhl, A.: Textual Views in Model Driven Engineering. In: SEAA 2009, pp. 133–140. IEEE, New York (2009)
11. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: A Framework for the Development of Textual Domain Specific Languages Categories and Subject Descriptors. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE Companion 2008, pp. 925–926. ACM, New York (2011)
12. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.: Bidirectional Transformations "bx" (Dagstuhl Seminar 11031). In: Dagstuhl Reports, vol. 1, pp. 42–67. Dagstuhl Publishing, Dagstuhl (2011)
13. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) OOPSLA 2010, pp. 444–463. ACM, New York (2010)
14. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
15. Lauder, M., Schlereth, M., Rose, S., Schürr, A.: Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. Bulletin of the Polish Academy of Sciences: Technical Sciences 58(3), 409–421 (2010)
16. Nagl, M.: Building Tightly Integrated Software Development Environments: The IPSEN Approach. Springer, Berlin (1996)
17. Parr, T.J.: Enforcing Strict Model-View Separation in Template Engines. In: Feldman, S., Uretsky, M. (eds.) WWW 2004, pp. 224–233. ACM, New York (2004)
18. Parr, T.J.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Lewisville (2007)
19. Parr, T.J.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. The Pragmatic Bookshelf, Lewisville (2009)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2009)
21. Stürmer, I., Kreuz, I., Schäfer, W., Schürr, A.: The MATE Approach: Enhanced Simulink and Stateflow Model Transformation. In: MAC 2007. MathWorks, Natick (2007)