Lessons Learned from Building a Graph Transformation System

Gabor Karsai

Institute for Software-Integrated Systems, Vanderbilt University,
Nashville, TN 37205, USA
gabor.karsai@vanderbilt.edu

Abstract. Model-driven software development is a language- and transformation-based paradigm, where the various development tasks of engineers are cast in this framework. During the past decade we have developed, evolved, and applied in practical projects a manifestation of this principle through a suite of tools we call the Model-Integrated Computing suite. Graph transformations are fundamental to this environment and tools for constructing model translators, for the specification of the semantics of languages, for the evolution of modeling languages, models, and their transformations have been built. Designing and building these tools have taught us interesting lessons about graph transformation techniques, language engineering, scalability and abstractions, pragmatic semantics, verification, and evolutionary changes in tools and designs. In the paper we briefly summarize the techniques and tools we have developed and used, and highlight our experience in constructing and using them.

Keywords: model-driven development, integrated development environments, graph transformations, model transformations.

1 Introduction

Model-driven software development is a language- and transformation-based paradigm, where the various development tasks of engineers are cast in this framework [35]. Models are used in every stage of the software product's lifecycle and the model-oriented thinking about the product permeates every aspect of the software engineer's work. Models are used to capture requirements and designs, assist in the implementation, verification, testing, deployment, maintenance, and evolution.

As there is no single language or tool that solves all these problems in software production no single modeling language or modeling tool can solve them either - hence a multitude of models is needed. Models are the artifacts of software production, and there are dependencies among these models: some models are closely related to each other (e.g. design models to requirement models), while some models (and other, non-model artifacts) are automatically generated from models. Two examples for the latter include 'analysis models' that are suitable for verification in some automated analysis tool (e.g. SMV) and executable code (e.g. in C); both of them are derived from the same source model (e.g. UML State Machines).

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 202–223, 2010.
© Springer-Verlag Berlin Heidelberg 2010

For a model-driven development process model transformations are essential: these transformations connect the various models and other artifacts, and they need to be executed frequently by the developers, or by some automated toolchain. Hence, constructing model transformation tools is of great importance for the builders of development toolchains. The model transformations have to be correct, reliable, robust, and provide high performance; otherwise the productivity of developers is reduced.

The advantages of using domain-specific approaches to software development and modeling are well recognized [1]. Using domain-specific modeling languages necessitates the development of custom, domain-specific model transformations – that are subject to the same quality requirements as any other transformations in a toolchain.

In the past 15+ years, our team has created and evolved a tool-suite for model-driven software development that we call 'Model-Integrated Computing' (MIC) suite [24]. The toolsuite is special in the sense that emphasizes (and encourages) the use of domain-specific models (and thus modeling languages), as opposed to focusing on a single general purpose approach (like UML). Hence model transformations (and especially domain-specific model transformations) play an essential role in the MIC suite. Another specialty of the suite is that it is a 'meta-toolsuite' as it allows defining and constructing domain-specific toolchains with dedicated domain-specific modeling languages.

The development of the MIC toolsuite involved creating the technology for all aspects of a domain-specific model-driven toolchain, including language definition (including concrete and abstract syntax, as well a semantics), model editing, specifying model transformations, the verification of models and model transformations, code generation, the evolution of models and model transformations. In this paper we focus on the model transformation aspects of the toolsuite and present what interesting lessons have been learned about graph transformation techniques, language engineering, scalability and abstractions, pragmatic semantics, verification, and evolutionary changes in tools and designs. In the text we will *indicate important lessons using the mark* [L].

The paper is organized as follows. First, we discuss the fundamental concepts related domain-specific modeling languages. Next, the main ideas used in model transformations are introduced; followed by the discussion on four selected problem domains: efficiency, practical use of transformations, verification of transformations, and the role of transformations in evolution and adaptation. The paper concludes with a summary and topics for further research.

2 Foundations: Metamodels

The first problem in constructing a domain-specific model-driven toolchain one faces is the specification and definition of domain-specific modeling languages (DSML) [24]. Formally, a DSML L is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S, and M_C):

$$L = < C, A, S, M_S, M_C >$$

The *concrete syntax* (C) defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The *abstract syntax* (A)

defines the *concepts, relationships,* and *well-formedness constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently identified as "static semantics". The *semantic domain* (*S)* is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The mapping M_C: $A \rightarrow C$ assigns concrete syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The semantic mapping M_S: $A \rightarrow S$ relates syntactic constructs to those of the semantic domain. The definition of the (DSM) language proceeds by constructing metamodels of the language (to cover A and C), and by constructing a metamodel for the semantics (to cover M_C and M_S).

One key aspect of the model-driven development (and in particular, MIC) is that *model-driven concepts should be recursively applied* [L]. This means that one should use models (and modeling languages) to define the DSMLs, and the transformations on those languages, and thus models should be used not only in the (domain-specific) work, but also in the engineering of the development tool suite itself. In other words, models should drive the construction of the tools. This recursive application of the model-driven paradigm leads to a unifying approach, where the tools are built using the same principles and techniques as the (domain) applications. Furthermore, one can create generic, domain-independent tools that could be customized (via models) to become domain-specific tools, in support of domain-specific models.

Models that define DSML-s are *metamodels*, and thus a metamodeling approach should support the definition of the concrete and abstract syntax, as well as the two mappings mentioned above. Obviously, the metamodels have a language, with an abstract and concrete syntax, etc. and this language is *recursively* defined, using itself. Thus, the metamodeling language is defined by a metamodel, in the metamodeling language – thus closing the recursion.

Through experience we have learned that *the primary issue one has to address in defining a DSML is that of the abstract syntax* [L]. It is not surprising, as abstract syntax is closely related to database schemas, conceptual maps, and alike that specify the core concepts, relationships and attributes of systems. Note that the abstract syntax imposes the inherent organizational principles of the domain and all other ingredients of a DSML are related to it.

We have chosen the concrete syntax of UML class diagrams to define the abstract syntax, as it is widely known, well-documented, and sufficiently precise. *When choosing a concrete syntax for a DSML it is important to use one that is familiar to the domain engineers* [L], in this particular case the language developers. A UML class diagram defines a conceptual organization for the domain, but also the data structures that can hold the domain models. This mapping from the class diagrams to data (class) structures has been implemented in many systems.

As discussed above, the definition of abstract syntax must include the specification of well-formedness rules for the models. We have chosen the well-documented OCL approach here: OCL constraints could be attached to the metamodel elements and they constrain the domain models. Note the difference: in conventional UML constraints restrict the object instances; here *the meta-level DSML constraints restrict the models (which are, in effect, instances of the classes of the abstract syntax)* [L].

One important issue with constraints is when and how they are used. As our constraints refer to the domain models, they are evaluated when those domain models are constructed and manipulated. At the time when domain models are constructed the modeler can invoke a 'constraint checker' that verifies whether the domain models comply with the well-formedness specified in the metamodel. Occasionally these checks are automatically triggered by specific editing operations, but most often the modeler has to invoke them. Often the checks are made just before a model transformation is executed on the models. *As these checks may involve complex computations, it is an interesting research question when exactly to activate them* [L]; after every editing operation, upon a specific modeler command, when the models are transformed, etc.

The *concrete syntax* defines the rendering of the domain model in some textual or graphical form. Obviously, the *abstract syntax can be rendered in many different concrete forms* [L], and different forms could be effective for different purposes. A purely diagrammatic form is effective for human observation, but an XML form is better for automated processing. *Choosing a concrete syntax has a major impact on the usability of a DSML* [L].

There have been a number of successful research efforts to make the concrete syntax highly flexible [17][37]. These techniques typically provide a (frequently declarative) specification for rendering the abstract syntax into concrete syntax as well as interpreting elementary editing events as specific operations that manipulate the underlying data structures of models. Alternatively, one can generate diagram editors from specifications [35]. These techniques are very flexible and general and can be used to create very sophisticated model editing environments. In effect, these techniques *operationalize* the mapping M_C above.

We have chosen a different approach that is less flexible but allows rapid experimentation with DSML-s; we call this 'idiomatic specification of concrete syntax' [25]. In our previous work, we have created a number of graphical modeling environments (graphical model editors) that have used a few model organization principles coupled together with a few visual rendering and direct manipulation techniques. For example, hierarchical containment, simple associations between model elements, associations between elements of disparate models that are contained within higher order models, and indirect referencing are such model organization principles that could be visualized using hierarchical diagrams, edges between icons, edges between ports of icons, and icons that act as pointers to distant model elements, respectively. Each such model organization principle is represented with a *visual idiom*. In our metamodeling language, each metamodel element has a stereotype that indicates the visual idiom to be used when rendering the corresponding domain model element. This approach, while much more limited than the approaches to relating concrete syntax to abstract syntax mentioned above, gives a rapid feedback for the designer of a DSML: the designer constructs UML class diagrams using predefined classes and associations with predefined stereotypes (e.g. <<Atom>>, <<Model>>, <<Connection>>, <<Reference>>, <<Set>>, etc.) and the resulting diagram immediately specifies not only the abstract syntax of the DSML, but also the concrete syntax. With the help of a generic visual modeling environment, one can experiment with the new DSML literally within seconds. This experience has shown that *choosing a simple*

technique for specifying the visualization of models could be very effective, although much less general than a full realization of the mapping M_C: $A \rightarrow C$ [L].

Defining the *semantics* of a domain-specific modeling language is a complex problem, and it is the subject of active research. In our MIC suite we have chosen a transformation based approach that we will discuss in a subsequent section.

Related work: Model-driven development is outlined in the Model-Driven Architecture vision of OMG, and it is an active area of research as illustrated by the series of conferences on 'Model-Driven Engineering Languages and Systems'. However, software engineering environments for model-driven systems development have evolved from classical integrated development environments [32], and many of the same problems appear (and are re-solved) in a newer setting. Arguably, the novelty in MDA/MDE is the use of codified modeling languages for object-oriented design (UML) and the increasing use of domain-specific abstractions and dedicated, visual, domain specific modeling languages [26]. As such, the focus in the model-driven approach is moving away from the classical (textual) 'document' oriented approach towards to (graphical) 'model' oriented approach. This has an implication on how the development artifacts are stored and manipulated: in classical text-oriented environments parsing and un-parsing are important steps, while in model-driven environments (graphical) models are often rendered graphically and manipulated directly. Interestingly, one can draw parallels between the data model for abstract syntax trees (for source code) and the metamodels: they capture the concepts, relationships, and attributes of a 'language' (for programming and modeling, respectively).

3 Model Transformations

Model transformations play an essential role in any model-driven development toolchain, as discussed above [35]. They integrate different tools, they are used in refactoring and evolving model-based designs, they were used to specify code generators, and they are used in everyday work, for rapid development activities. Additionally, their efficiency, quality, and robustness are of great importance for pragmatic reasons: inefficient transformations lengthen development iterations, poor quality transformations produce inefficient models or code, and brittle transformations can cause great frustrations among developers.

It is widely recognized in the model transformation community that graph transformations serve as a suitable foundation for building model transformation systems. Graph transformations are not the only approach, but because of their long history and solid mathematical foundations they provide a solid background upon which model transformation systems can be built.

3.1 Model Transformations via Efficient Graph Transformations

Graph transformations are specified in the form of graph rewriting rules, where each rule contains a left-hand-side graph (LHS) and right-hand-side graph (RHS) [39]. When a rule is applied, an isomorphic occurrence of the LHS in the input graph is

sought and when found it is replaced by the RHS of the rule. Typically a transformation consists of more than one rule and these are applied in some order, according to some specification (either implicit or explicit). Note that the input and the output of the transformation are typed graphs, where each node has a specific type, and a rule matches only if node types match between the LHS (the pattern) and the input (host) graph.

Graph rewriting rules offer a very high-level formalism for defining transformation steps. It is easy to see that the procedural code that performs the same function as a rewriting rule could be quite complex. In fact, every graph rewriting rule execution involves a subgraph isomorphism search, followed by the manipulation of the target (output) graph. *The efficiency of the graph transformation is thus highly dependent on the efficiency of the graph isomorphism search* [L].

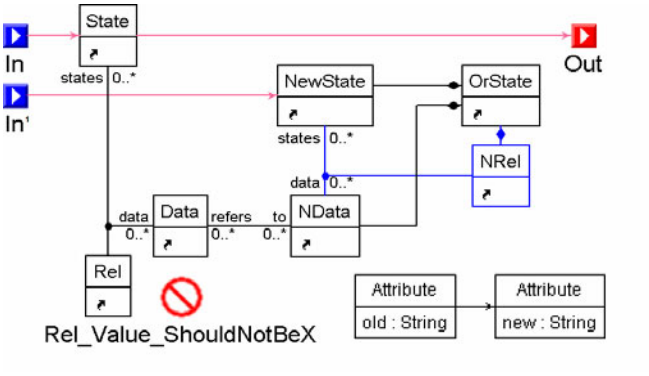

Fig. 1. Model transformation rule example

The worst-case run-time complexity of graph isomorphism test is exponential in the size of the graphs involved, but in graph transformations we only search for a fixed pattern, and the worst case time complexity for is $O(n^k)$, where n is the size of the graph and k is the size of the pattern. In our graph transformation-based model transformation system, called GReAT [2], we further reduced this by using localized search. The host graph is typically much larger than the pattern graph, and the pattern matches in a local neighborhood of a relative small number of nodes. Such localized search can be achieved by pre-binding some of the nodes in the pattern to specific nodes in the host graph. As shown on Fig. 1, the `State` and `NewState` nodes of the pattern are bound to two nodes in the graph (`In` and `In`^{1}, respectively), before the rest of the pattern is matched. In other words, the pattern is not matched against all nodes in the host graph, rather only in the neighborhood of selected, specific nodes. When the rule is evaluated, the pattern matcher produces a set of bindings for the pattern nodes `Rel`, `Data`, `NData`, and `OrState`, given the fixed binding of the nodes `State` and `NewState`. Starting the search from 'pivot' nodes leads to significant reduction in the complexity of the pattern matching process as the size of the local context is typically small (provided one avoids the so-called V-structures [11]).

Localized search, however, necessitates the determination of the locale, i.e. the binding of the `State` and `NewState` nodes in the example. This problem was solved by recognizing the need for a traversal path in the input graph. General purpose graph transformation approaches perform graph matching on the entire graph, and this has serious implications on the execution speed of such systems. In our system, similarly to some other systems like PROGRES [41] and Fujaba [25], we require the developer to explicitly provide a traversal path that sets how the rewriting rules are applied in the input graph. In practical systems model graphs have a well-known 'root' node where traversal can start form, and the first step in the transformation must have a rule that binds that root to one of the pattern nodes. A rewriting rule can also 'hand over' a node (existing, i.e. matched or newly created) to a subsequent rule (this is indicated on the example by the connection from the `State` to the `Out` port of the rule). Note that the patterns expressed in the rewriting rules and the sequencing of the rules (i.e. connecting the output ports of rules to input ports of other rules) implicitly specify how the input graph is traversed (and thus how the rewriting operations are sequenced). The sequencing can be combined with a hierarchy, as shown on Fig. 2.

While the approach appears to be more complex (and 'lower-level' compared to general graph transforms), in practice it is quite manageable. Depth-first and breadth-first traversals, traversals using arbitrary edge types, even fixpoint iterations over the graph are straightforward to implement. In our experience, *trading off generality and developer's effort for efficiency in the resulting transformation results in transformations that are not only reasonably fast (on large graphs) but also easier to understand, debug, and verify* [L].

In our research, we wanted to build 'industrial strength' model transformations that operate on large models. Our first implementation of the model transformation engine was completely 'interpreted': the engine executed the rewriting rule sequence on the input graph; exhibiting expected performance shortcomings. Once the semantics of the transformation rules and programs was clear and stabilized, we have developed a code generator that produced executable code from the transformation models. The generator was implemented using the well-known partial evaluator technique and it produced code based on the partial evaluation of the transformation program with respect to the transformation interpreter semantics. *For practical applications, such a 'compilation-based' approach to enhancing the performance of model transformations is essential* [L].

Related work: PROGRES [41] is arguably the first widely used tool for specifying transformations on structures represented as graphs. PROGRES has sophisticated control structures for controlling the rewriting the process, in GReAT we have used a similar, yet different approach: explicitly sequenced rules that form control flow diagrams. PROGRES also supports representing type systems for graphs; in GReAT we use UML diagrams for this purpose. The very high-level notion of graph transformations used in PROGRES necessitates sophisticated techniques for efficient graph matching ([10] [39]); in GReAT we mitigate this problem by using less powerful rules and (effectively) performing a local search in the host graph.

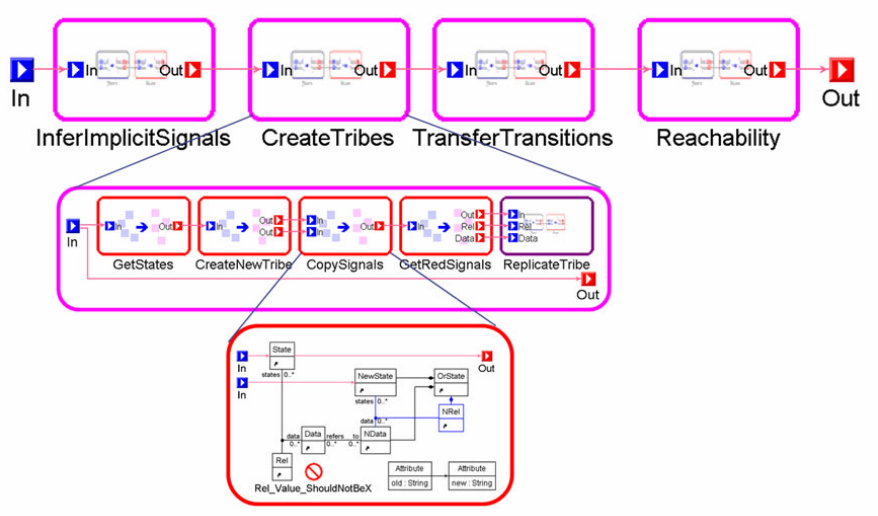

Fig. 2. Sequencing and hierarchy of rewriting rules

Fujaba [25] is similar to GREAT in the sense that it relies on UML (the tool was primarily built for transforming UML models) and uses a technique for explicitly sequencing transformation operations. Fujaba follows the state-machine-like "story diagram" approach [13] for scheduling the rewriting operations; a difference from GReAT.

AGG [43] is a graph transformation tool that relies on the use of type graphs, similar to UML diagrams but does not support all UML features (e.g. association classes). Recent work related to AGG introduced a method for handling inheritance, as well as a sophisticated technique for checking for confluence (critical pair analysis). In GReAT, inheritance is handled in the pattern matching process, and the confluence problem is avoided by using explicit rule sequencing. AGG has support for controlling the parsing process of a given graph in the form of layered grammars; a problem solved differently in GReAT.

VIATRA [5] is yet another graph transformation tool with interesting capabilities for controlling the transformations (state machines), and the composition of more complex transformations. In GReAT similar problems were addressed via the explicit control flow across rules and the formulation of blocks. Higher-order transformations were also introduced in VIATRA; there is no similar capability in GReAT currently.

GReAT can also be compared to the recent QVT specification [39] of the OMG MDA standardization process. However, we should emphasize that GReAT was meant to be a research tool and not an industry standard. With respect to the QVT, the biggest difference between GReAT and QVT is in the highly declarative nature of the QVT: it focuses on relation mappings. This is a very high-level approach, and it is far from the pragmatic, lower-level, efficiency-oriented approach followed in GReAT. We conjecture that describing a transformation in QVT is probably more compact, but the same transformation in GReAT is more efficient.

In a more general setting, we should compare GReAT and the tool environment it belongs to, i.e. the MIC tools including GME and UDM. Honeywell's DOME [13], MetaCASE's MetaEdit [25], and the ATOM3 [28] environment are the most relevant examples that support domain-driven development. The main difference between them and the MIC tools is in the use of UML and OCL for metamodeling and the way the metamodels are (also) used for instantiating a visual modeling environment. Also, our transformations follow a high-level method for describing the transformation steps expressed in the context of the metamodels. With exception of ATOM3, all the above tools use a scripting language, in contrast.

3.2 Practical Use of Model Transformations

The model transformation environment we have created has been used in a number of academic and practical projects. Students, researchers, and developers have used it to create practically useful transformations ranging from converting between modeling formalisms to generating code from models. Some of the transformations were of the 'once-only' (or 'throw-away') type; some of them are in daily use in tools. In these efforts we have learned a few important lessons discussed below.

1. **Reusable patterns.** *Given a model transformation language, developers often discover important and useful transformation patterns that are worth documenting and reusing* [L]. These patterns are essentially generic transformation algorithms that are usable across a number of domain specific modeling languages. Conceptually, they are like C++ template libraries that provide generic data structures and algorithms over domain-specific types. Practically, they provide a reusable solution to a recurring transformation problem. Such patterns are rarely implemented by a single rule, rather, by a sequence or group of rules.

   To increase the reusability of such transformation patterns, a model transformation language should support templates [L], which are rules or rule sequences that are parameterized with types. When the transformation designer wishes to use a transformation template, s/he can bind the type parameters to concrete, domain-specific model element types and the tool environment should instantiate the pattern.
2. **Cross-domain links.** In model transformations the source and target models typically (but not always) belong to different metamodels (i.e. different type systems). During the transformation process it is often necessary to create a link between two model elements that belong to different domains (metamodels), but this brings up the question: which metamodel does the association belong to? Neither of the source or target metamodels 'owns' such an association, the association belongs to the cross-product space of the two. Hence, *the model transformation system should be able to allow such 'cross-domain' links* [L]; at least temporarily, while the transformation is being executed [2].
3. **Global context.** The localized rewriting approach described above has a practical shortcoming: the context of the rewriting has to be always present during the execution of a rule [41]. That is, a rule cannot just create an 'orphan' target model element – the element has to be inserted into an appropriate container, which is in the target context. In other words, the state of the transformation

often has to be incrementally built and passed from rule to rule. This leads to rules that have superfluous input and output ports, just for passing the context through; and a large number of connections between rules. *The simplicity of the transformation model is important, and such 'accidental complexity' confuses developers* [L]. We have introduced the concept of a 'global container,' where new temporary model elements can be created and latter found via search. In other words, a rule can create a model element in this container and a subsequent rule can refer to it simply referring to the container and using the pattern matcher to find the element. *Note that such global containers are useful, although somewhat 'unhygienic' tools for implementing model transformations* [L].

4. **Multiple matches and sorting.** A model transformation rule often matches to multiple isomorphic subgraphs in an input graph, and even a localized rewriting rule could generate multiple, consistent matches with different bindings to pattern nodes. In general, the order of such matches is nondeterministic as it depends on how the underlying 'model database' is implemented. We chose to process these matches sequentially, and for every match we execute the right hand side of the rule, which leads to non-deterministic results. We found that in many applications *the order of processing of such matches does matter* [L]. To solve this problem we have introduced an optional 'sort' function for the rewriting rule that the designer can specify [41]. This function is applied to the result of the pattern matcher before the rule is actually executed, and the function can sort the results in any order of interest. *How the matches need to be sorted is domain-specific, hence it is better left in the hands of the developer* [L].
5. **Multiple matches and grouping.** When the pattern matcher generates a collection of matches (each one with a distinct set of bindings of input graph nodes to pattern nodes), the rewriting rule processes them one by one. *The major limitation with this simple algorithm is the inability to apply a single rule action across multiple matches* [L]. After all matches are computed, the rule's action (RHS) is executed individually, on each match; furthermore, there exists no mechanism by which one can access information about an earlier match while processing a specific match. This can sometimes pose a severe limitation to the types of transformations one can write. For instance, the user may need the ability to operate on an entire subgraph (composed from multiple matches) as a whole rather than on individual elements. If this subgraph may contain an arbitrary number of elements, then the graph pattern cannot be specified as a simple rule.

   We have introduced a higher-order 'subgroup' operator that allows forming groups from the matches during rule execution [3] [4]. The operator has a number of attributes the designer can specify, including functions that are evaluated to determine whether a match belongs to a group or not. The operator extends the rule execution semantics as follows: (1) the pattern matcher produces a set of matches; (2) matches are used to form groups, based on functions supplied with the operator, (3) the rule is executed for each group formed. Note that a group may contain one or more matches. *The subgroup operator has demonstrated the value of higher-order operators in rewriting rules that can operate across multiple individual rewriting steps* [L].

The above extensions have come up in practical transformation problems, and they showed that while graph transformations provide a powerful theory, when applied to model transformations they need to be specialized and adapted to pragmatic goals.

3.3 The Issue of Verification

The quality of a model-driven software development toolchain is determined by the quality of the tools it includes and the model transformations that connect together these tools. Considering that elements in the toolchain could support verification (on the code or on the model level), the *verification of the transformations, i.e. graph transformations, is of great importance* [L]. Simply, the correctness of the transformation is necessary in order to decide that the result of an automated verification applies to the original input (model), and to the code generated from the model.

Our goal was to decide the correctness of a model transformation through some verification process. The verification of model transformations is closely related to the verification of compilers – one of the great challenges of computer science. Arguably, the verification of model transformations is simpler, as the domain-specific modeling languages are often simpler and have a simpler semantics than general purpose programming languages.

One important observation is that the notion of correctness is not absolute, but it has to be defined with respect to some specific domain property, which is of interest to the users of the toolchain. For example, a model transformation can be called correct with respect to the behaviors generated by the source and the target models. For instance, the transformation is correct if the source model (with its own source semantic domain) generates the same behaviors as the target model (with its own target semantic domain). Alternatively, a transformation is correct when a property of the source model holds if and only if another property holds for the target model.

Practical model transformations are often very complex and the formal proof of correctness requires a major effort. Note that a formal proof shows that the given model transformation is correct (w.r.t. some property), for *any* input. Another feasible approach is that the proof is constructed for a particular *run* (or 'instance') of the transformation, i.e. for a given input. This, *instance-based verification of the transformation appears to be much simpler and feasible* [L]. While the concept has been developed in the context of program generators [5], we have successfully applied it to model transformations [13].

Constructing the verification for a transformation instance requires building a tool that checks what the transformation did and verifies it independently. These checks must be simple and easily verifiable. Note that this concept is similar to provers and proof checkers: the proof checking is typically much simpler than constructing the proof. For a model transformation one needs (*1) to choose the property the correctness is defined for, (2) to discover how this property can be verified from data collected during the run of the transformation, (3) to modify the model transformation to generate the data during the run, and (4) to develop (and verify) the algorithm that checks the data and thus verifies the property* [L].

One example for such transformation and verification property includes a transformation between two transition system formalisms and a state reachability property. In this case the transformation needs to generate a map that links source and target states, and the checking algorithm must verify that there is a correspondence: a strong bisimilarity between the two transition systems, hence reachability properties verified for one do hold for the other [13].

Related work: The MOF 2.0 Query / View / Transformation specification [39] provides a language for declaratively specifying transformations as a set of relations that must hold between models. A relation is defined by two or more domains, and is declared either as *Checkonly*, meaning that the relation is only checked, or *Enforced*, meaning that the model is modified if necessary to satisfy the relation. It is augmented by a *when* clause that specifies under what conditions the relation must hold, and a where clause that specifies a condition that must be satisfied by all the model elements participating in the relation. Our approach provides a solution similar to the *Checkonly* mode of QVT relations. The main difference is our use of pivot nodes to define correspondence conditions and the use of cross links. This allows us to use a look up table to match corresponding nodes. Our approach takes advantage of the transformation framework to provide a pragmatic and usable verification technique that can ensure that there are no critical errors in model instances produced by automated transformations. Triple Graph Grammars [41] can be used to represent the evolution of a model graph by applying graph transformation rules. The evolving graph must comply with a graph schema at all times. This graph schema consists of three parts, one describing the source metamodel, one describing the target metamodel, and one describing a correspondence metamodel which keeps track of correspondences between the other two metamodels. Triple graph grammar rules are declarative, and operational graph grammar rules must be derived from them. The correspondence metamodel can be used to perform a function similar to the cross links used here. This provides a framework in which a map of corresponding nodes in the instance models can be maintained, and on which the correspondence conditions can be checked. This makes it suitable for our verification approach to be applied. Some ideas on validating model transformations are presented in [28] and [29]. In [28], the authors present a concept of rule-based model transformations with control conditions, and provide a set of criteria to ensure termination and confluence. In [29], the authors focus on the syntactic correctness of rule-based model transformations. This validates whether the source and target parts of the transformation rule are syntactically correct with respect to the abstract syntax of the source and target languages. These approaches are concerned with the functional behavior and syntactic correctness of the model transformation. We focus on the semantic correctness of model transformations, addressing errors introduced due to loss or misrepresentation of information during a transformation. It is possible for a transformation to execute completely and produce an output model that satisfies all syntactic rules, but which may still not have accomplished the desired result of porting essential information from the

source model to the output model. Our approach is directed at preventing such semantic errors. Ehrig et. al. [13] study bidirectional transformations as an approach for preserving information across model transformations. They use triple graph grammars to define bidirectional model transformations, which can be inverted without specifying a new transformation. Our approach offers a more relaxed framework, which will allow some loss of information (such as by abstraction), and concentrates on the crucial properties of interest. We also feel that our approach is better suited for transformations involving multiple models and attribute manipulations.

3.4 Transformations in Evolution and Adaptation

One of the crucial properties of software systems is the need for their evolution and adaptation. Software must evolve, as new requirements arise, as flaws need to be fixed, and as the system must grow in its capabilities. *Model-driven development toolchains are also software systems, and hence the same requirement applies: they need to evolve and adapt* [L]. The problem is especially acute for tools that use domain-specific modeling languages, as the DSML-s may evolve not only from project to project, but often during the lifetime of one project.

The issue of evolution for a DSML is not only how the language changes, but also what effect this has on the already existing models. Specifically, if a large number of models have already been built with a DSML of version *n*, how do we use these models with DSML version *n+1*, etc.? The problem is related to schema evolution (i.e. how we evolve database content when the schema evolves), but modeling languages typically have much richer semantics and consistency constraints than typical database schemata. *For DSML-s the language evolution problem is essentially a model migration problem, i.e. how to migrate models when the DSML evolves* [L].

The problem can be cast as a model transformation problem, i.e. how can one create model transformations that automatically migrate the models from one DSML version to the next. To analyze this problem we need to recall how a DSML is defined; i.e. the metamodels. In this paradigm, the DSML evolves via changes applied to the metamodels; i.e. changes in the abstract and concrete syntax, in the well-formedness constraints, and in the semantic mapping. As model transformations are anchored in the abstract syntax of the DSML it is natural to consider the metamodel changes on that level. Changes in the concrete syntax do not affect the models (until a syntax-free realization of the models exists), while changes in the well-formedness constraints and semantic mapping could possibly be also formulated as a model transformation problem. These latter two cases could be formulated as posing the question: how shall the models be transformed that they comply with the updated well-formedness constraints (if at all) and how they shall be transformed such that they preserve their semantics under the updated semantic mapping?

Changes in the abstract syntax part of the metamodel involve changing the UML class diagrams representing that. Such changes can be captured as elementary editing operations on the diagram, including adding, removing, and modifying classes and associations, etc. But focusing on these low-level changes makes it exceedingly hard to discover the (meta-) modeler's intent. For instance removing a class called `Failure` and adding a class `Fault` may miss the point that this is a simple renaming of a class

without changing the semantics. Hence, *evolution in the abstract syntax cannot be dependably deduced by observing editing changes on the metamodels; the modeler needs to provide guidance or explanations for such changes* [L]. For pragmatic reasons, naturally, only the changes need to be documented (or discovered by some automation, if feasible) – *parts of the metamodels that did not change should not become subject to model transformations* [L].

Such analysis lead us to a simple (graphical) language that allows the modeler to document the metamodel changes by capturing how 'old' metamodel elements are related to 'new' metamodel elements [32]. Note that the modeler essentially supplies rewriting rules that proscribe how a model migration engine should convert 'old' models into 'new' models. In this graphical language, called Model Change Language (MCL), we have defined a number of idioms for representing prototypical cases for metamodel changes (and thus model migration). Fig. 3 illustrates the major idioms of the language – these have been discovered through practical experience with model migration problems. While the use of these idioms has been proven useful in specific model migration problems, the formal semantics of MCL is subject of active research. Naturally, model evolution is not a solved problem yet, but transformations appear to offer interesting opportunities.

A migration rule specified in MCL describes a migration step which is centered on a single, specific 'old' metamodel element that dominates the rule. The semantics of the migration rule is as follows: whenever the dominant model element is found in the 'old' model and the left-hand side of the rule matches; then execute the migration as specified (e.g. create a 'new' model element, etc.). If an 'old' model element is encountered that is not mapped by a migration rule then check if there is a 'new' metamodel element with the same name and create that in the 'new' model, if there is none then give a warning that an 'old' model element was encountered but could not be mapped. Note that there are explicit rules for saying that some 'old' model elements need to be removed because there are no corresponding 'new' model elements – this allows detecting that the migration of some model elements was not specified correctly by migration rules.

In MCL we faced the problem of limiting the scope of the search and we found a solution that appears to work well. The solution is based on the observation that *model databases mostly follow a tree structure, and a dominant spanning tree can be found for the model graph* [L], often via the model containment hierarchy. Hence, we first use a depth-first traversal on the model tree, visiting every node in the graph and trying to find a migration rule. The rule semantics briefly described above is applied, when possible. However, there could be rules that are not applicable yet, because they depend on model elements that have not been visited and processed yet. These rules are pushed onto a queue of delayed rules and the traversal continues. Once the depth first traversal terminates, we keep processing the delay queue until it becomes empty. This simple, fixed traversal strategy works surprisingly well. Arguably, *for practical model-driven systems that use hierarchical organization model transformations can efficiently be performed using a depth-first traversal, followed by the processing of rewriting steps that had to be delayed during the first traversal* [L].

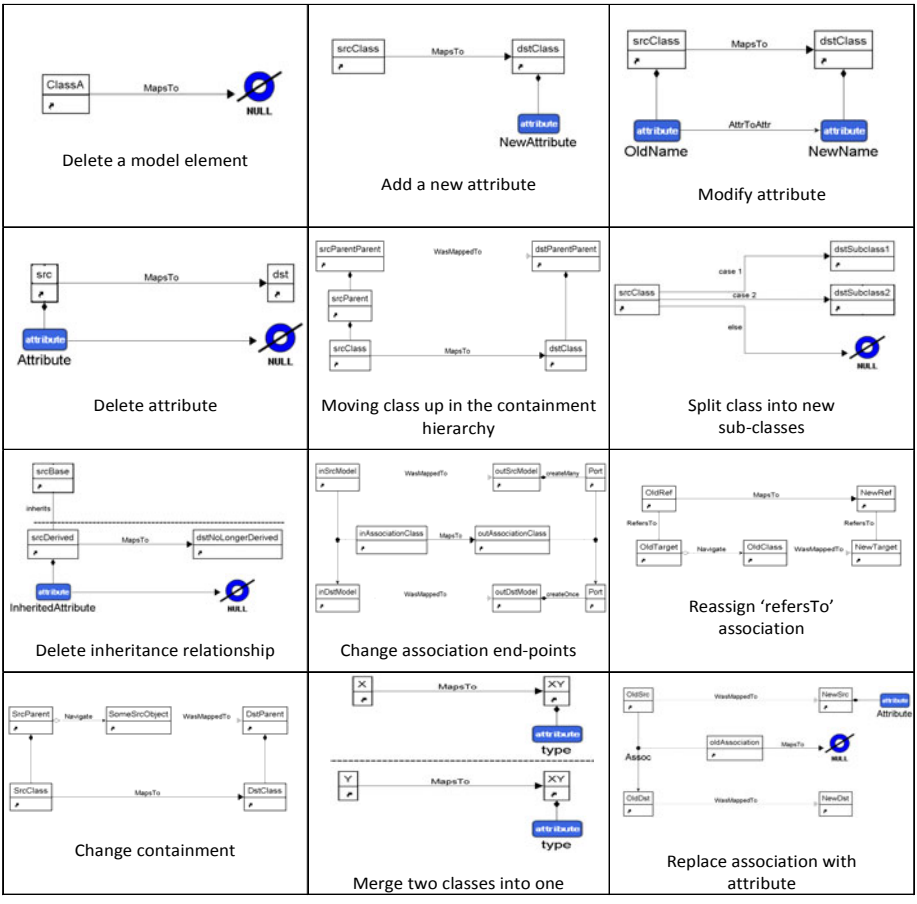

Fig. 3. Some idioms of the Model Change Language

Related work: Our work on model migration has its origins in techniques for database schema evolution [5]. More recently, though, even traditional programming language evolution has been shown to share many features of model migration. Drawing from experience in very large scale software evolution, [15] uses several examples to establish analogies between tradition programming language evolution and metamodel and model co-evolution. Using two industrial metamodels to analyze the types of common changes that occur during metamodel evolution, [17] gives a list of four major requirements that a model migration tool must fulfill in order to be considered effective: (1) Reuse of migration knowledge, (2) Expressive, custom migrations, (3) Modularity, and (4) Maintaining migration history. The first, reusing migration knowledge is accomplished by the main MCL algorithm: metamodel independent changes are automatically deduced and migration code is automatically generated. Expressive, custom migrations are accomplished in MCL by (1) using the metamodels directly to describe

the changes, and (2) allowing the user to write domain-specific code with a well-defined API. Our MCL tool also meets the last two requirements of [17]: MCL is modular in the sense that the specification of one migration rule does not affect other migration rules, and the history of the metamodel changes is persistent and available to facilitate model migration. [8] performs model migration by first examining a difference model that records the evolution of the metamodel, and then producing ATL code that performs the model migration. Their tool uses the difference model to derive two model transformations in ATL: one for automatically resolvable changes, and one for unresolvable changes. They note that the generated transformation that deals with the unresolvable changes must be refined by the user, but details of how to accomplish this refinement are not provided. Also, [7] does not specify exactly how the difference models are calculated, only that they can be obtained by using a tool such as `EMFCompare`. MCL, on the other hand, uses a difference model explicitly defined by the user, and uses its core algorithm to automatically deduce and resolve the breaking resolvable changes. Changes classified as breaking and unresolvable are also specified directly in the difference model, which makes dealing with unresolvable changes straightforward: the user defines a migration rule using a graphical notation that incorporates the two versions of the metamodel and uses a domain-specific C++ API for tasks such as querying and setting attribute values. In [7], the user has to refine ATL transformation rules directly in order to deal with unresolvable changes. [17] describes the benefits of using a comparison algorithm for automatically detecting the changes between two versions of a metamodel, but says they cannot use this approach because they use `ECore`-based metamodels, which do not support unique identifiers, a feature needed by their approach. Rather than have the changes between metamodel versions defined explicitly by the user, they slightly modify the `ChangeRecorder` facility in the EMF tool set and use this to capture the changes as the user edits the metamodel. Their migration tool then generates a model migration in the Epsilon Transformation Language (ETL). In the case that there are metamodel changes other than renaming, user written code in ETL to facilitate these changes cannot currently be linked with the ETL code generated by their migration tool. In contrast to this, MCL allows the user to define complex migration rules with a straightforward graphical syntax, and then generates migration code to handle these rules and links it with the code produced by the main MCL algorithm. [10] presents a language called COPE that allows a model migration to be decomposed into modular pieces. They note that because metamodel changes are often small, using endogenous model transformation techniques (i.e., the metamodels of the input and output models of the transformation are exactly the same) can be beneficial, even though the two metamodels are not identical in the general model migration problem. This use of endogenous techniques to provide a default migration rule for elements that do not change between metamodel versions is exactly what is done in the core MCL algorithm. However, in [19], the metamodel changes must be specified programmatically, as opposed to MCL, in which the metamodel changes are defined using a straightforward graphical syntax. Rather than manually changing metamodels, the work in [45] proposes the use of QVT relations for evolving metamodels and raises the issue of combining this with a method for co-adapting models. While this is an interesting idea, our MCL language uses an explicit change language to describe metamodel changes rather than model transformations.

Evolution of a DSML (and the subsequent migration of domain models) is not the only activity the toolchain users face. They often need to evolve, adapt their designs by changing models. In object-oriented software development perhaps the most powerful concept for design adaptation is the use of design patterns. By definition, design pattern is a general reusable solution to a commonly occurring problem in software design. When developers use a design pattern they modify their designs according to the pattern, in other words they instantiate the design pattern in the context of their work. If the design is captured in a model, then a design pattern is a particular arrangement of newly built or existing model elements, possibly with some new features added. Arguably, *a design pattern can be modeled as a specialized model transformation rule that rewrites a design into a new design with the design pattern features (model elements, attributes, etc.) added* [L]. Note also that design patterns applied in domain-specific modeling languages will have domain-specific elements hence they can be called as 'domain-specific design patterns'.

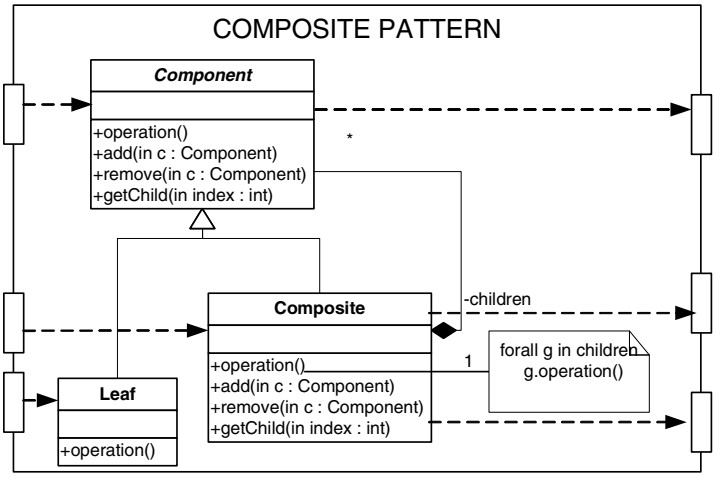

Fig. 4. The Composite Pattern as a model transformation rule

One example for realizing a design pattern as a transformation rule is shown on Fig. 4. Here the well-known 'Composite' design pattern is used. When the designer wants to introduce this pattern into a design, s/he will either just copy it into the model and modify it, or bind it to existing model elements (say, Compound and Primitive, shown on the left) which will result in a modified model that contains the original as well as new elements (shown on Fig. 5).

Note that the application of design patterns becomes an interactive activity this way that the modeler performs at model construction time. Design patterns can be applied to existing design, and they can extend or even refactor those designs. Design patterns can be highly domain specific hence they can be applied in any DSML.

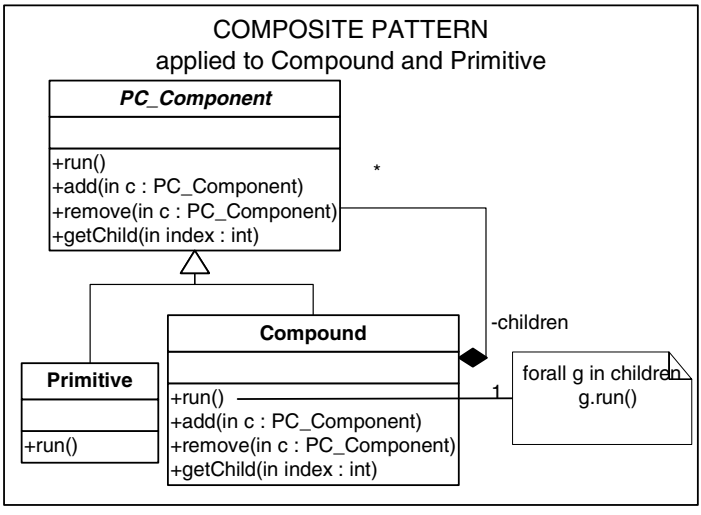

Fig. 5. Composite pattern applied

We have created a set of tools to support the definition and application of design patterns in arbitrary DSML-s that are defined by a metamodel [32]. One tool is used (once) to extend the metamodel of a given DSML such the patterns can be built from existing model elements. Another tool is available to the modeler that uses the DSML: this tool applies the pattern as a local model transformation. The modeler can bind existing model elements to elements of the pattern and the applicator tool extends and modifies the model as specified by the pattern.

Related work: There are several mainstream tools that support UML design patterns, or describe design patterns using general-purpose languages, as opposed to using the metamodel of the DSMLs. Moreover, there are several approaches for pattern formalization. Here, we reference the closest related work only. Previous work [32] has justified the demand for Domain-Specific Model Patterns by contributing several DSMLs. Moreover, it describes relaxation conditions for the metamodels in order to make metamodeling environments support the editing of incomplete models. As opposed to the approach introduce above, it deals with static model patterns only. In our approach, relaxations can be made on the metamodel of the pattern environment. The multiplicities can be substituted with the upper bound of the multiplicity set, dangling edges can be defined with ignored end nodes and transitive containment can be solved with ignored containers. Incomplete attributes can be implemented the same way. [17] describes a UML-based language, namely, the Role-Based Metamodeling Language (RBML), which is able to specify domain-specific design patterns. This approach treats domain patterns as templates, where the parameters are roles, and a tool generates models from this language. Compared to our approach, the paper [22] proposes a formal way to specify the pattern embedding for the static aspect. The behavioral formalization is closely coupled with design patterns defined in UML. The work described in [5] formalizes the embedding, tracing, and synchronization between several

pattern aspects that may be defined in different languages. These results constitute an excellent theoretical formalization of the tracing aspects for model patterns defined in the static aspect.

4 Summary and Conclusions

The model-driven development approach has significantly changed how software is built and evolved, and new development environments are coming equipped with model-driven support. The techniques and the tools we have developed in the past decade indicate that model-driven development works, but the complexity of the development tools (and the effort to build them) is increasing as well. In this paper we have outlined a few of the lessons that we have learned during building and using our tools on non-trivial projects. Building tools to build software is essential to solve the software development problem and the effort put into constructing a good tool (-suite) pays off in developer's productivity. The lessons described in this paper show steps in an evolutionary process, and by no means should be considered the final word on model-driven development. As tools and techniques evolve, we need to learn new lessons, and enable the developers to benefit from them.

Acknowledgements. This work was sponsored, in part, by the Evolutionary Design of Complex Systems and Software, the Model-based Integration of Embedded Systems, and the Software Producibility programs of DARPA and AFRL, and by the NSF ITR on "Foundations of Hybrid and Embedded Software Systems". The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA, NSF, or the US government. The work described in this paper has involved a number of researchers, engineers and students at the institution of the author, including but not limited to: Janos Sztipanovits, who started it all, Aditya Agrawal, Arpad Bakay, Daniel Balasubramanian, Jeff Gray, Zsolt Kalmar, Akos Ledeczi, Tihamer Levendovszky, Endre Magyari, Miklos Maroti, Anantha Narayanan, Benjamin Ness, Sandeep Neema, Feng Shi, Jonathan Sprinkle, Ryan Thibodeaux, Attila Vizhanyo, Peter Volgyesi. All this is really their work.

References

1. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G.: Reusable Idioms and Patterns in Graph Transformation Languages. Electronic Notes in Theoretical Computer Science 127, 181–192 (2005)
2. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal on Software and System Modeling 5, 261–288 (2006)
3. Balasubramanian, D., Narayanan, A., Neema, S., Shi, F., Thibodeaux, R., Karsai, G.: A Subgraph Operator for Graph Transformation Languages. ECEASST 6 (2007)
4. Balasubramanian, D., Narayanan, A., Neema, S., Ness, B., Shi, F., Thibodeaux, R., Karsai, G.: Applying a Grouping Operator in Model Transformations. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 410–425. Springer, Heidelberg (2008)

5. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In: Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data, pp. 311–322 (2007)
6. Bottoni, P., Guerra, E., Lara, J.: Formal Foundation for Pattern-Based Modelling. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 278–293. Springer, Heidelberg (2009)
7. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC, pp. 222–231 (2008)
8. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: VIATRA — Visual Automated Transformations for Formal Verification and Validation of UML Models. In: IEEE Conference on Automated Software Engineering, pp. 267–270 (2002)
9. Denney, E., Fischer, B.: Certifiable Program Generation. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 17–28. Springer, Heidelberg (2005)
10. Dörr, H.: Efficient Graph Rewriting and its implementation. LNCS, vol. 922. Springer, Heidelberg (1995)
11. Dörr, H.: Bypass Strong V-Structures and Find an Isomorphic Labelled Subgraph in Linear Time. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 305–318. Springer, Heidelberg (1995)
12. DSLs: The Good, the Bad, and the Ugly, Panel at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Nashville, TN, October 22 (2008)
13. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Fundamental Approaches to Software Engineering, pp. 72–86 (2007)
14. Engstrom, E., Krueger, J.: Building and rapidly evolving domain-specific tools with DOME. In: IEEE International Symposium on Computer-Aided Control System Design, pp. 83–88 (2000)
15. Favre, J.M.: Meta-models and Models Co-Evolution in the 3D Software Space. In: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICS (2003)
16. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
17. Fondement, F., Baar, T.: Making metamodels aware of concrete syntax. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 190–204. Springer, Heidelberg (2005)
18. Gruschko, B., Kolovos, D.S., Paige, R.F.: Towards Synchronizing Models with Evolving Metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution (MODSE) (2007)
19. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE: A Language for the Coupled Evolution of Metamodels and Models. In: MCCM Workshop at MoDELS (2009)
20. Herrmannsdoerfer, M., Benz, S., Jurgens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
21. Kim, D.-K., France, R., Ghosh, S., Song, E.: A UML-Based Metamodeling Language to Specify Design Patterns. In: Proc. Workshop Software Model. Eng. (WiSME) (2004)

22. Kim, S.-K., Carrington, D.A.: A formalism to describe design patterns based on role concepts. Formal Asp. Comput. 21(5), 397–420 (2009)
23. Karsai, G., Narayanan, A.: On the Correctness of Model Transformations in the Development of Embedded Systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop 2006. LNCS, vol. 4888, pp. 1–18. Springer, Heidelberg (2007)
24. Karsai, G., Ledeczi, A., Neema, S., Sztipanovits, J.: The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design. In: IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany (2006)
25. Kelly, S., Tolvanen, J.-P.: Visual domain-specific modelling: Benefits and experiences of using metaCASE tools. In: Bezivin, J., Ernst, J. (eds.) Proceedings of International Workshop on Model Engineering, ECOOP 2001 (2000)
26. Kent, S.: Model Driven Engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods, May 15-18, pp. 286–298 (2002)
27. Klein, T., Nickel, U., Niere, J., Zündorf, A.: From UML to Java And Back Again, Tech. Rep. tr-ri-00-216, University of Paderborn, Paderborn, Germany (September 1999)
28. Küster, J.M.: Systematic validation of model transformations. In: Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004) (October 2004)
29. Küster, J.M., Heckel, R., Engels, G.: Defining and validating transformations of uml models. In: HCC 2003: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments, Washington, DC, USA, pp. 145–152. IEEE Computer Society, Los Alamitos (2003)
30. de Lara, J., Vangheluwe, H.: AToM3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
31. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer, 44–51 (November 2001)
32. Levendovszky, T., Lengyel, L., Mészáros, T.: Supporting domain-specific model patterns with metamodeling. Software and Systems Modeling (March 2009), doi:10.1007/s10270-009-0118-3
33. Levendovszky, T., Karsai, G.: An Active Pattern Infrastructure for Domain-Specific Languages. Accepted for presentation at First International Workshop on Visual Formalisms for Patterns (VFfP 2009), Corvallis, Oregon, USA (2009)
34. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996) ISBN 3-540-61985-2
35. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Sci. Comput. Program. 44(2), 157–180 (2002), http://dx.doi.org/, doi:10.1016/S0167-6423(02)00037-0
36. Model-Driven Architecture Guide, OMG, http://www.omg.org/docs/omg/03-06-01.pdf
37. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schnekenburger, R., Gérard, S., Jézéquel, J.M.: Model-driven analysis and synthesis of concrete syntax. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 98–110. Springer, Heidelberg (2006)
38. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic Domain Model Migration to Manage Metamodel Evolution. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 706–711. Springer, Heidelberg (2009), http://dx.doi.org/, doi:10.1007/978-3-642-04425-0_57
39. OMG QVT specification, http://www.omg.org/docs/ptc/05-11-01.pdf

40. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Publishing Co. Pte. Ltd., Singapore (1997)
41. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
42. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRES. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 219–234. Springer, Heidelberg (1995)
43. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
44. Vizhanyo, A., Neema, S., Shi, F., Balasubramanian, D., Karsai, G.: Improving the Usability of a Graph Transformation Language. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), March 27, 2006. Electronic Notes in Theoretical Computer Science, vol. 152, pp. 207–222 (2005)
45. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
46. Zündorf, A.: Graph pattern matching in PROGRES. In: Cuny, J., Engels, G., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 454–468. Springer, Heidelberg (1996)