Noname manuscript No.
(will be inserted by the editor)

Lessons Learned from Building Model-Driven Development Tools

Richard F. Paige[1] **and Dániel Varró**[2]

Department of Computer Science,
University of York, UK.
e-mail: `paige@cs.york.ac.uk`
Department of Measurement and Information Systems
Budapest University of Technology and Economics, Hungary.
e-mail: `varro@mit.bme.hu`

The date of receipt and acceptance will be inserted by the editor

Abstract Tools to support modelling in system and software engineering are widespread, and have reached a degree of maturity where their use and availability are accepted. Tools to support *Model-Driven Development (MDD)* – where models are manipulated and managed throughout the system/software engineering lifecycle – have, over the last ten years, seen much research and development attention. Over the last ten years, we have had significant experience in the design, development and deployment of MDD tools in practical settings. In this paper, we distill some of the important lessons we have learned in developing and deploying two MDD tools: Epsilon and VIATRA. In doing so, we aim to identify some of the key principles of developing successful MDD tools, as well as some hints of the pitfalls and risks.

1 Introduction

The last ten years have seen the development of numerous tools for supporting Model-Driven Development (MDD): the manipulation and analysis of structured descriptions. MDD tools such as ATL [23], xText [42], Epsilon [16], VIATRA [5], ATOM3 [15], KerMeta [29], Acceleo [1], FUJABA [13], MOFLON [3], GReAT [2] and many others have been developed and deployed in a variety of software and systems engineering contexts. Many of these tools are the result of research and development on specific industrial use cases; others are the result of theoretical and conceptual research. Some are now being applied on large-scale software engineering projects and are being turned into products. As such, the field of MDD tool development has reached a sufficient level of maturity for its results to be assessed, and distillation of lessons learned from the development of these tools can take place.

This paper aims to identify some of the principles underpinning the design, implementation and evolution of MDD tools. The principles will be distilled from an analysis of the development of two MDD tools that are used in practice: VIATRA and Epsilon. These tools are used in industry, on real projects, and have developed in very different ways. Arguably, some of the lessons learned from the development of these tools can inform the development of new MDD tools, and can also be used to support the evolution of existing tools. Our objective is not to propose the "ideal" MDD tool; such a tool is unlikely to exist. However, the principles underpinning the development of MDD tools that are widely used in practice can help in assessing existing tools, and in improving them.

To elicit and present the lessons learned, the paper will summarise the development of Epsilon and VIATRA, starting from initial use cases and system requirements, leading to an initial tool architecture. Evolutionary steps (e.g., refactorings) will be discussed and presented, along with the triggers – which included both requests for new functionality, as well as requests related to non-functional characteristics such as performance and scalability – that led to these refactorings.

The paper is structured as follows. Section 2 describes the development and evolution of Epsilon and VIATRA, focusing on initial motivation and scenarios, as well as some of the most important evolutionary steps. Section 3 distills some of the key take-home messages from these developments. We conclude in Section 4.

2 Evolution of MDD Tools

This section presents the development and evolution of two MDD tools: Epsilon and VIATRA. Both Epsilon and VIATRA are predominantly textual tools, featuring textual languages for specifying and executing MDD tasks. Each part describes the tool as it currently exists, then talks about its development cycle. The development of Epsilon starts with an initial use case, and then moves to a discussion of refactoring stages, including both technical refactorings (i.e., to improve performance) and conceptual refactorings. The development of VIATRA starts from model transformation requirements, including significant requirements for performance, and then broadens its scope to include other MDD tasks.

2.1 The design and evolution of Epsilon

2.1.1 Epsilon Epsilon [16] is a tool for model management, specifically for analysing and manipulating models in specific ways. More precisely, it is a set of textual *task-specific model management languages*, and a framework for implementing *new* model management languages by exploiting the existing ones. It is a component of the Eclipse Model Framework Technology (EMFT) project.

From a user's perspective, Epsilon provides a set of inter-related and largely interoperable languages that can be used to implement and execute various model management tasks. The current architecture of Epsilon is illustrated in Figure 1.

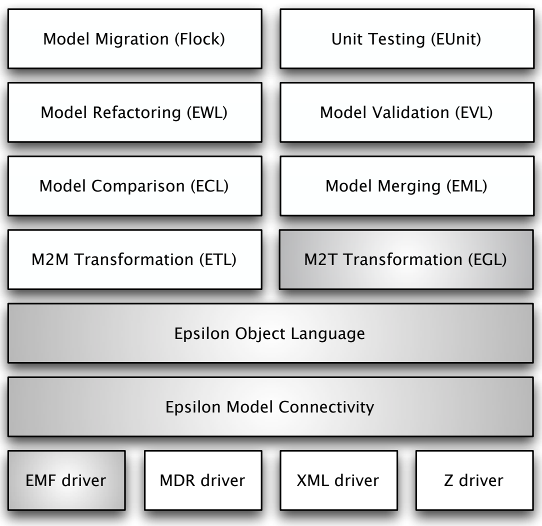

Fig. 1 Epsilon model management platform

Epsilon consists of a set of textual languages (each including parsers, editing tools, and interpreters/virtual machines), a connectivity framework (more on that later), and some additional tools to help ease development. Each language has further development tools (e.g., syntax-aware editors). Each language aims to support a particular model management task. More specifically, there is a language for direct manipulation of models (EOL) [26], as well as languages for model merging (EML) [24], model comparison (ECL) [25], model-to-model transformation (ETL) [27], model validation (EVL) [28], model-to-text transformation (EGL) [36], model migration (Flock) [35]) and unit testing of model management operations (EUnit).

The core language in Epsilon is EOL; all other languages in Epsilon reuse EOL in some way, as it provides the conceptually common features required for manipulating and analysing models. In particular, it supports navigation of models (via OCL-like expressions and queries), modification of models (via assignment statements), and multiple-model access capabilities. However, EOL was – at least conceptually – not the first language to be developed; it emerged from the development of a different Epsilon language, as we now discuss.

2.1.2 Initial use case and architecture The use case that triggered the development of Epsilon was one of *model merging*. Model merging is the process of combining two or more source models (possibly from different modelling languages) into a target model (which may conform to a completely different metamodel than the source models). The model merging scenarios that were of interest were two-fold:

- support for version control on models, including combining different versions of models, identifying differences, etc.
- support for merging *behavioural* models (e.g., state machines, sequence diagrams).

Our research suggested that the process of model merging could be separated into four stages:

1. A *comparison phase*, where correspondences between equivalent elements of source models are identified;
2. A *conformance checking phase*, where corresponding elements identified in the previous phase are examined to identify conflicts that may render merging impossible. This phase was particularly important for version control, i.e., when merging models of the same metamodel.
3. A *merging phase*, where corresponding and conforming elements are combined; and
4. A *reconciliation phase*, where any inconsistencies introduced in the merging phase are resolved.

What is revealing about this separation is that each phase – comparison, 'checking', merging – is an operation that we might want to apply to models. Indeed, it was this observation that suggested that model merging itself was a *composite* operation that can be applied to models – that is, merging was constructed from other model management operations.

This raised the question *what did model management operations have in common?* To this end, we investigated model transformation, merging, constraint checking (e.g., via OCL) and model-to-text transformation, and identified a small set of features that all these operations have in common:

- navigating models;
- accessing multiple models simultaneously;

– evaluating expressions on models;
– modifying models

We also noticed that different MDD tools tended to reimplement many of these common features using non-standard syntaxes, thus rendering interoperability of tools more difficult (in part because syntax transformation would have to take place) We argued [26] that there was substantial value in encapsulating these features so that they could be reused in different operations that required them. This was the genesis of the core language of Epsilon – EOL – and the conceptual architecture depicted in Figure 1.

In parallel with this conceptual process, we obtained (from our collaborators who provided the initial use case) constraints and requirements on the tools we were committed to produce. In particular, the collaborators required a textual interface to any tools (the intended users preferred a textual interface instead of a graphical one). It was also perceived that a textual interface, and textual MDD languages, were preferred for fine-grained tasks such as specifying how models were navigated, evaluating expressions, etc. This was unsurprising and parallels the development of formal specification languages, as well as model management langauges such as OCL. At this time we also made a decision to implement our MDD tools in an interpretive style (as opposed to a code generator style), in order to quickly build a proof-of-concept, and to allow us more flexibility in evolving the tools quickly in the future. Since we were not unduly concerned with performance at this stage, this seemed a judicious decision.

Tools for EOL were then defined and implemented (i.e., parsers, editors, an interpreter). These were developed largely without following an MDD approach – we exploited *grammarware* technology and well-understood concepts from the compiler and interpreter community. This was because our focus was on rapidly developing novel MDD tools (based around the conceptual analysis of model management operations described earlier) and supporting industrial MDD use cases, instead of 'eating our own dog food' and developing MDD tools following an MDD approach.

Based on the tools for EOL, we then implemented a merging language – EML [24] – to support the scenarios mentioned earlier. EML, a rule-based merging language, supported the phases of merging mentioned earlier, while reusing the core features of EOL to support navigation and multiple-model access. However, even at this stage we identified redundancy in the core features of EML. We discuss this in the next section.

The intent with the design of EOL, from the start, was for it to be reused in defining and implementing other model management operations. At this stage, we had identified model-to-text transformation as a representative example, as well as refactoring; these will be discussed in a following section. Before that, however, we discuss a conceptual refactoring.

2.1.3 Conceptual refactoring After designing and implement EOL and EML, and fulfilling our initial requirements and use cases, we took a step back and studied the core features of EML. In particular, we observed two recurring patterns in the *logic* of the merging programs that we were writing in EML:

– Significant parts of the EML programs focused on writing the expressions that would compare the models. In many places, these expressions would be repeated. A reusable *operation* concept was added to EOL to simplify EML programs, but even with this there was still substantial opportunities for reuse of *comparison logic* that was not supported by EOL (or EML).
– Some of the rules in the EML programs simply transformed elements from one source language into the target language. Effectively, some EML rules were just *transformation* rules.

These observations led to the first major conceptual refactoring of Epsilon: from a language consisting of a core set of features (EOL) and a merging language, to a platform of languages including EOL and EML, as well as a language for transformation (ETL).

Effectively, the syntax and semantics of EML informed the design of the transformation language, ETL. We studied the patterns of transformation that were expressed in EML, and designed a syntax (and an execution engine) that would efficiently and concisely support these patterns. This in turn simplified the design and structure of EML; no longer would EML programs have to express transformation logic as well as merging logic.

In parallel with the development of ETL, and the refactoring of EML, we investigated the comparison logic inherent in EML programs in more detail. As noted above, we identified many situations where comparison operations were expressed – artificially – using EOL, as well as unexploited opportunities for reuse. As a result, we defined a new model management like, ECL, to support the definition of comparison operations. As with ETL and EML, ECL reused EOL to provide basic features such as model navigation and expression definition. The definition of ECL further simplified the definition of EML. Now EML reused ETL for transformation rules, ECL for identifying correspondences between models, and only defined the logic of model merging itself.

This was a substantial *conceptual* refactoring, which had a significant impact on end-users of Epsilon. In terms of the underlying infrastructure of Epsilon there was less of an impact, in part because much of the functionality needed to support ETL and ECL already existed.

2.1.4 Infrastructural refactoring At this stage in Epsilon's evolution, the platform had a significant number of users interested in transformation, merging (version control, specifically) and model comparison. As the number of users increased, requests for improved performance and for handling larger models became more prevalent.

The collection of requests for improved performance led to a number of infrastructural refactorings:

– Because of requirements from industry partners, Epsilon from the start supported models of arbitrary types, including EMF/Ecore, MDR (MOF) and pure XML. This was provided via a connectivity layer that abstracted Epsilon programs (e.g., in EML) from the technology used to store the models. This meant that Epsilon users could write programs without having to be concerned with their models' representations, but it also meant that executing a statement on a model incurred overhead (since the connectivity layer had to redirect calls to specific technology handlers). As a result, and also because of complex licensing issues (e.g., MDR/MOF is licensed differently from EMF/Ecore), tailored versions of Epsilon became available, particularly standalone versions for EMF/Ecore models, and MDR models. This improved efficiency significantly.
– Some users requested the ability to obtain more detailed, fine-grained information about the performance of specific parts of the Epsilon programs. A profiler was developed which supported this.
– Under the hood, Epsilon consists of a number of parsers (implemented using ANTLR) and a number of virtual machines that orchestrate the execution of Epsilon programs. In parallel with the refactoring of the connectivity layer, above, the way in which Epsilon virtual machines interact was simplified. There were two side-effects of this. Firstly, an explicit user-accessible way for orchestrating Epsilon programs was now needed, e.g., so that users could run a sequence of programs one after the other. ANT was used as a lightweight mechanism for this. Secondly, a number of patterns were identified for defining *new* Epsilon languages from existing ones; arguably these patterns became more apparent after simplifying the ways in which Epsilon languages interact at execution time.
– For a number of users, performance issues arose because of the way in which they had constructed their (very large) models: typically as a set of smaller models with cross-references. An analysis demonstrated that maintaining these cross-references was particularly expensive as the models changed. As a result, the infrastructure of Epsilon evolved to support a new framework for cross-model references, called Concordance [34], targeted specifically at this problem.

This period of infrastructure consolidation and refactoring took approximately a year (though some of the research results were published much later), and in our view spending significant effort on improving the performance of the infrastructure paid off later, particularly in terms of retaining users for whom performance was a significant concern.

An issue that arose in this period was the integration of Epsilon with other modelling tools (e.g., for creating UML models) and more general-purpose software engineering tools (e.g., compilers, debuggers, code profilers). Integration with modelling tools – such as Rational Rose or MagicDraw – was carried out loosely, via interoperability at the modelling technology level, e.g., in terms of XMI, XML, or other persistent format. Integration with software engineering tools was also carried out loosely via a persistent format (e.g., generated code) and through use of the aforementioned ANT tasks. As a result the development of Epsilon largely avoided some of the difficult issues – e.g., complex API interpretation – associated with interoperating with external tools.

2.1.5 New tasks After this period of work on Epsilon's infrastructure, we entered a phase of experimentation with new tasks and MDD scenarios. A number of new languages, supporting new operations, emerged from this phase, including the model-to-text language EGL [36], an inter-model consistency checking and validation language, EVL, an update-in-place transformation language, EWL, a model migration language, Flock, and a number of development tools, including EuGENia, an application of Epsilon for generating the models required for GMF editors.

The stages by which these languages developed had certain commonalities:

– Typically, we were given or obtained an MDD scenario, e.g., refactoring a model.
– We attempted to implement the scenario using existing Epsilon languages or tools. EOL, for example, is computationally complete and is sufficient for implementing any MDD task; using EOL is not necessarily the most suitable approach, as the resulting program may be complicated, cumbersome, or difficult to reuse or maintain.
– The attempt to use an existing Epsilon language for solving an MDD problem sometimes did not succeed; sometimes the results were unsatisfactory (e.g., less maintainable). In these situations, as explained in [30], we attempted to identify the recurring logical patterns in the Epsilon code we had written, to give us requirements for a *new* Epsilon language.
– We defined and implemented an abstract and concrete syntax for the new Epsilon language (reusing, where possible, existing syntaxes and machines). A number of *language patterns* of reuse were identified from this, and are explained in detail in [30].

Through this general process, EVL, EWL and Flock were all developed. EGL (the model-to-text language) was developed differently: the requirements for model-to-text transformation were not supported by any existing Epsilon language, and as such we did not carry out significant experiments with EOL (or other languages) before implementing EGL.

By this stage in the evolution of Epsilon, there was a clear meta-pattern: each language was logically coherent, supporting one task (e.g., comparison, model-to-model transformation, evolution, etc). Languages that overlapped with others, thus leading to duplication in functionality, were to be refactored and simplified, and those interactions between languages made *explicit*, at the programmer level, through use of the ANT workflow. This key idea continues to influence the development of Epsilon to this day.

A reasonable question to ask – and, indeed, we did ask it during the stages of development of new Epsilon languages – was whether it would have been better to develop Epsilon as a single, general-purpose language (e.g., just EOL) for model management, instead of a family of small interoperable languages. We do not yet have a clear answer to this question yet, but evidence so far suggests that there is likely a minimum level of *complexity* to an MDD task that would benefit from the development of a task-specific language (as opposed to use of a general-purpose language). An example of this arose with the development of Flock, Epsilon's model migration language. Migration (like other MDD tasks) can be implemented and carried out with EOL directly. However, Flock was developed after some experiments with using EOL (and other Epsilon languages) demonstrated that it was at the wrong level of abstraction, and substantial amounts of boilerplate code had to be written to accomplish a conceptually simple task. A simple migration task implemented in Flock may take only a few lines of code, whereas the same task in EOL may take many dozens of lines of code (if not more). This observation was made for many of the tasks supported by Epsilon's languages, and while not conclusively indicating that task-specific languages were always more compact, at least provided some preliminary evidence that they could be more compact than general-purpose languages for the same problems.

2.2 The design and evolution of VIATRA

2.2.1 The VIATRA model transformation framework The main objective of the VIATRA2 (VIsual Automated model TRAnsformations) framework [5,38] is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains.

VIATRA2 primarily aims at designing model transformations to support precise model-based systems development with the help of invisible formal methods. Invisible formal methods are hidden by automated model transformations projecting system models into various mathematical domains (and, preferably, vice versa). The VIATRA2 model transformation framework is available as an official (open source) Eclipse Generative Modeling Tools (GMT) subproject [41].

The main specialities of VIATRA2, when compared with other MDD tools, include

– a model space for hierarchical and uniform representation of large models and metamodels based upon VPM metamodeling [39]
– transformation language with both declarative and imperative features based upon popular formal mathematical techniques of graph transformation (GT) and abstract state machines (ASM) [38]
– a high performance transformation engine supporting (1) incremental model transformations, (2) event-driven live transformations [31] where complex model changes may trigger execution of transformations, and (3) handling well over 1,000,000 model elements [6]
– with main target application domains in model transformations for model-based tool integration [4] and model-based analysis [17].

Target application domains The most traditional application area for VIATRA2 transformations – starting as early as 1998 – is to support the transformation-based dependability analysis of system models [12] taken from various application areas (safety-critical and/or embedded systems, robust e-business applications, middleware, service oriented architecture) described using various modeling languages (BPM, UML, etc.) during a model-driven systems engineering process. Such a dependability analysis typically also includes the verification & validation, the testing, the safety and security analysis as well as the early assessment of non-functional characteristics (such as reliability, availability, responsiveness, throughput, etc.) of the system under design. In addition, model transformations for specification, design, deployment, optimization or code generation in traditional model-driven systems engineering are also focal areas for VIATRA2.

Target audience and end users The VIATRA2 framework served as the underlying model transformation technology of many European projects in the field of dependable embedded systems and service-oriented applications. In this way, academic and industrial partners in these projects became the first end users of the framework. Regular usage of the framework has been reported at ARCS and TU Vienna (Austria), University of Leicester (UK), LMU Munich (Germany), TU Kaiserslautern (Germany), University of Pisa (Italy), Georgia University of Technology (USA) and University of Waterloo (Canada). The VIATRA2 framework also serves as the foundation of an industrial design toolkit for automotive systems developed at OptXware Ltd. for the AUTOSAR architecture.

2.2.2 The evolution of VIATRA

The first version of VIATRA The first version of the VIATRA model transformation framework [14] was developed between 2000 and 2004. Source and target metamodels and models were exported and imported in XMI 1.0 format. Transformation rules were captured in an off-the-shelf UML tool using a dedicated UML profile for model transformations. These transformation rules were translated into a Prolog representation to be executed by the SWI Prolog engine after manually writing some glue code in Prolog for controlling the transformation process. While this first Prolog version of VIATRA already offered acceptable performance for most transformations, transformation development was a "one-man show" due to lack of knowledge of Prolog.

The framework was completely rewritten from scratch in Java in 2004 based upon Eclipse. Since then, the system continuously improved along three major and several minor releases with contributions from 7 PhD students, several MSc students, and some employees of OptXware, a spin-off company of the research group.

Back in 2004, our main design goals were to develop a framework that

1. was usable by an average software engineer (or at least by own MSc students),
2. integrated well with existing modeling languages (e.g. UML, SysML, BPEL) and technologies (e.g. Java, XML)
3. used a semantically well-founded transformation language strongly related to formal notations like graph transformation and abstract state machines, and
4. offered significantly better runtime performance compared to graph/model transformation tools that existed in 2004.

Architectural overview of VIATRA2 The overall architecture of VIATRA2 is summarized in Fig. 2. Models, metamodels and transformations are stored in the VPM model space, where all these artefacts can be manipulated via the core model management interfaces. While entire model spaces (with related models and transformations) can be loaded and saved in an XML based generic representation, in practice, model spaces in VIATRA2 are typically populated by using importers adapted to native formats of various languages and tools or using the VIATRA Textual Modeling Language (VTML). After model transformation programs (specified in VTCL files) are parsed and loaded to a model space, they can be executed on selected input models (in principle, on arbitrary model elements from the model space). Then the efficient execution of the transformation is carried out by the graph pattern matcher and the respective interpreter components for graph transformation (GT) and abstract state machine (ASM) rules.

Having a high emphasis on performance, we did not use our transformation language for developing VIATRA2 as a software tool. But as transformation models were used from very early days, certain optimization steps could potentially be reimplemented within the framework.

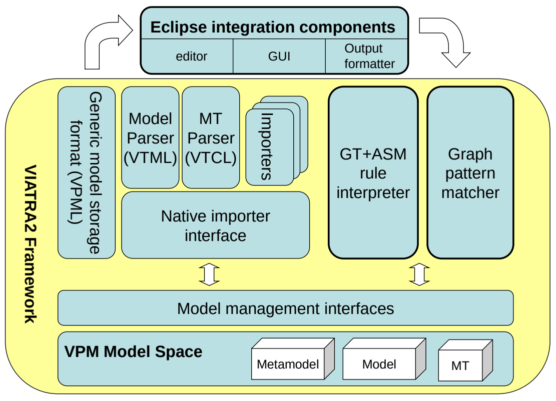

Fig. 2 The VIATRA2 model transformation framework

How did VIATRA2 succeed? As our foundational research has primarily been directed to improve the performance of the framework by proposing advanced graph pattern matching techniques based upon local-search [22, 40] or incremental techniques [6, 7], VIATRA2 scores very well among model transformation tools from a performance aspect. In addition, new model transformation concepts (like incremental graph transformation [6], live/event-driven transformations [31], or change-driven transformations [10]) were successfully integrated to VIATRA2 on the tooling level as well.

VIATRA2 also proved to be successful from a tool integration point of view. Importers and exporters have been developed (in research and student projects) to popular standardized modeling languages (like UML, BPEL, SysML), back-end formal analysis tools (SAL, PEPA, CPLEX, etc.) and a wide range of deployment platforms of industrial relevance (e.g. ARINC 653, TTA, Apache Axis2, WSDL) mainly in the field of critical embedded systems and service-oriented computing. Some of these scenarios (like [10]) included significant interoperability challenges where the target model was not materialized, but only a target API was available with limited traceability information.

While the use of VPM as a (non-standard) underlying metamodeling and model management paradigm definitely improved the expressiveness of transformations, we had to invest more for developing exporters and importers for these languages and tools compared to an EMF-compliant model transformation tool.

Choosing a semantically well-founded transformation language by combining two formal paradigms had advantages and disadvantages. On the positive side, the graph pattern language of VIATRA2 became very popular (not only among our students but also among collaborating researchers and software engineers at our spin-off company) by offering a succinct, reusable and very expressive language for capturing model queries and constraints (especially, when compared to OCL). On the negative side, only very few students actively used the declarative notation of graph transformation — they used graph patterns for the query part, but followed a more imperative transformation style for expressing the effects of a rule using abstract state machine constructs.

Despite continuous efforts to improve usability, the entry level of writing a first model transformation in VIATRA2 has remained relatively high (e.g. compared to ATL). Most external VIATRA2 users received some consultation from members of our research group in the initial phases of their transformation development. After learning the system, they were typically able to carry out their model transformation task, but this still indicates that we should have invested more in providing better documentation, tutorials, and improving the overall usability of VIATRA2.

Descendents of VIATRA2 The evolution of VIATRA2 in the recent years have also been dominated by the introduction of new components, which used model transformations in the background (as a service), but provided additional support for model driven design.

- **Domain-specific modeling over VIATRA2** The ViatraDSM framework [33] investigated a non-generative approach for designing rich domain-specific modeling environments, which extensively used model transformations for synchronization between abstract and concrete syntax [32] or for capturing discrete event based simulators [33].
- **Design space exploration over VIATRA2** Support for model-driven design space exploration [18, 20] provides a guided search using permitted operations to reach a designated goal state. It uses the transformation language of VIATRA2 to capture the design problem and it heavily relies upon incremental transformations to improve the efficiency of state space traversal.
- **Incremental query evaluation for EMF models** The main goal of the EMF-IncQuery framework [8, 9] is to provide efficient incremental query evaluation for EMF models. On the tooling level, EMF-IncQuery adapts the graph pattern language of VIATRA2 [11] to capture model queries, and the incremental pattern matcher of VIATRA2 to obtain high performance. EMF-IncQuery can be integrated as a validation service for almost arbitrary existing EMF based applications.

While VIATRA2 offers an interpreter to execute the transformation rules, this is by no means the only meaningful architectural choice. In fact, the EMF-IncQuery framework provides both an interpreted and compiled mode for supporting incremental model queries. Our experience is that the flexibility of the interpreted mode is highly beneficial for designing and testing the queries while the compiled mode is helpful in integrating queries to existing applications.

2.2.3 Industrial experience and feedback on model transformations Using the VIATRA2 model transformation framework and its descendents in industrial (and industry driven research) projects in an avionics and automotive context (at our research group and also at our spin-off company) gave valuable feedback for us, which guided several recent research directions.

For instance, the need for validating complex constraints over EMF models (i.e. EMF-IncQuery as a research project) arose from our spin-off company when developing automotive modeling tools. As the AUTOSAR standard contains several hundreds of well-formness constraints, the software engineers of the company had difficulties in providing an efficient manual implementation of these constraints.

We gained extensive industrial experience for model-driven configuration design both in context of automotive [19, 37] and avionics systems [21]. Here the underlying deployment platforms were substantially different (TTA, AUTOSAR or ARINC 653), the underlying design concepts turned out to be quite similar. While our initial attempts aimed at providing automated model transformations for as many design steps as possible, industrial domain experts suggested to decrease the level of automation by using user-guided model transformations aligned with precise development workflows [4]. Essentially, model transformations can assist designers in making the right design decision (by proposing design alternatives and calculating consequences of design decisions), but an automated model transformation is not allowed to make a design decision.

It was also interesting to notice the industrial priorities in this context. Developing rich graphical views of models and providing instantaneous support for early validation of design rules and guidelines were of higher priority compared to providing powerful graphical editors for models. Graphical editors were also beneficial in both an AUTOSAR and avionics context, but required significant programming effort.

3 Lessons Learned and Principles

In this section, we distill some of the key lessons learned from our years of experience in developing, maintaining and deploying both VIATRA and Epsilon. These lessons learned capture insights from both mistakes we have made as well as successes that we have had in solving specific and general problems.

To help to present the lessons learned systematically, we classify them as follows:

- *Requirements*: lessons learned about establishing, eliciting, specifying and negotiating requirements for MD-D tools, including lessons regarding end-users and stakeholders.
- *Architecture:* lessons learned about architectural principles and styles for MDD tools.
- *Modelling Technology:* lessons learned about the underlying modelling technology used to express models, and in particular the interfaces between the MDD tools and the modelling technology.
- *Environmental Constraints:* lessons learned about interfaces of the MDD tool to the wider systems engineering lifecycle.

3.1 Requirements

In the development of Epsilon and VIATRA, we learned several important lessons about setting and specifying requirements for MDD tools.

1. *Start from a rich MDD scenario*; it will help motivate and trigger the development of your MDD tool, and if it is sufficiently complex, it will guide your development for several iterations.
   The initial MDD scenario for Epsilon, based on model merging/composition, had essential complexity that was not immediately apparent to the Epsilon developers (including, for example, that comparison and transformation were also key tasks that supported model merging). This essential complexity introduced technical and conceptual challenges into the development of Epsilon that steered its development over the first two years. Had we started from a less complex scenario, the inherent flexibility and generality of the toolset may have been lessened, and the resulting tool may have been more specialised. This is not in itself a problem - a specialised MDD tool can be extremely valuable. However, the development of an MDD tool is in itself part of a scientific exploration; having the ability to conduct experiments while being *guided* – but not overly constrained – by initial scenarios is of significant value.
   In case of VIATRA, the main initial motivating scenario was related to mapping dynamic modeling languages (like statecharts, activity diagrams) to formal languages such as Petri nets or transition systems to carry out model-based analysis. This necessitated to specify model transformations bridging large abstraction gaps between source and target languages. As a result, VIATRA offered an expressive and rich transformation language right from the beginning.
2. *Try to have real end-users: they keep you honest.* This is important in general in systems engineering, but it is particularly important for MDD tools, which exist to *automate* tasks that end-users shouldn't be doing (i.e., rote, repetitive system engineering tasks). Both VIATRA and Epsilon had real end-users, with real requirements for automation, performability and reliability, from day one. This guided architectural design, technical design, and implementation and (because of the nature of the end-user engagements) meant that rapid feedback on new features and improved infrastructure could be obtained. Furthermore, the entire EMF-IncQuery project emerged from actual needs and requirements of potential end users.
3. *Start with complex challenges, transition to simpler challenges.* This is similar to the first lesson, but here we are focusing more on satisfying requirements than on identifying and using them to guide development. Both VIATRA and Epsilon started development with complex MDD challenges – e.g., model merging, fully recursive patterns, meta-transformations, generic transformations – which led to significant technical solutions, powerful infrastructure, and a very general understanding of what was difficult in MDD tooling and what was straightforward to support. Later iterations focused on simpler MDD challenges – e.g., code generation, incremental queries, interactive transformations – that made the technical solutions developed in earlier iterations easier to use and deploy.
   By focusing on complex issues to begin with, it was possible to identify components (e.g., comparison tool, generic transformation tool) in the technical solutions that may have had value as packaged, standalone tools. This was partly explored for both VIATRA and Epsilon.
4. *Worry less about the accuracy and suitability of the abstractions chosen, and more about how easy it is to change and extend those abstractions.* Establishing the requirements for an MDD tool is, in effect, a modelling problem. When developing both Epsilon and VIATRA, we spent considerable time trying to ensure that we were working at the most appropriate level of abstraction for the MDD task at hand, e.g., that the abstractions used for specifying model transformations or queries were accurate, justified and conceptually simple. It is obvious that the amount of time you can spend on assessing the fidelity and suitability of your abstractions is potentially infinite. Indeed, we could have experimented for years with different combinations of languages in Epsilon, building more capability in to the core technologies of both VIATRA and Epsilon, etc. Arguably, the most scientific, justfiable approach would be to agree on a justifiable and reasonable set of abstractions for an MDD task *that can be extended or changed easily*, and to experiment with these 'proposals' until they are either proven or shown to be invalid. This approach has generally guided the development of Epsilon and VIATRA in more recent times.

3.2 Architecture

We now outline some of the key lessons we have learned related to the architecture of MDD tools, in terms of evolving an architecture, and also establishing non-functional requirements via the chosen architecture.

1. *There is a significant design choice to be made early on (in developing initial versions of MDD tools) between building a more usable tool and one that performs better; focusing on one such property will help you to make progress.* This is unsurprising. However, the developments of both Epsilon and VIATRA focused on *one* of these extra-functional properties at the start of development; Epsilon focused on usability, while VIATRA focused on performance. At later stages of development, focus switched (e.g., from usability to performance in the case of Epsilon). This had advantages and disadvantages. In the case of VIATRA, focusing on performance meant that the MDD tool was capable of manipulating extremely large models efficiently from the start; this in turn meant that large modern standards (such as UML 2.x, AUTOSAR, MARTE) could be addressed directly with the tool without significant change. The disadvantage of this is that VIATRA's usability lagged behind its performance, and a significant learning curve was the result. In contrast, Epsilon's development focused on usability from the start – particularly because there were end-users who required the toolset by a specific deadline. This meant that the tool *always* had users (who provided feedback) who drove the development of new and novel features. However, Epsilon's performance wasn't a focus at the start; it never performed significantly worse than user expectations, but when attempts were made to manipulate extremely large models (on the order of megabytes), it did not perform acceptably. Later versions of both VIATRA and Epsilon have addressed concerns of usability and performance, respectively.
2. *Getting the architecture right from the start is less important than having a flexible architecture that can be refactored.* The architecture for both MDD tools has changed significantly over different versions. In both cases, we have been more concerned with having architectures that allow us to experiment with the features provided by the tools (e.g., adding new features, dropping features, splitting features). Tight coupling between the features of the tools has always been viewed as problematic, since it reduces flexibility and experimentation. For instance, significant re-engineering had to be dedicated to get rid of tight coupling between some VIATRA2 components introduced in early versions of the framework.
   Having a flexible architecture for an MDD tool is also important because the external standards (e.g., MOF, Ecore, UML, OCL) and tools (e.g., EMF, GMF, ANTLR) that the tools depend on will change; similarly, the licensing for external tools and standards may change, as well, and minimising the effect of said changes on the MDD tools is beneficial.
3. *Interpreters let you make progress quickly; code generators may be preferable for performance and integration.* We found that using interpreters (virtual machines) to support the languages in Epsilon allowed us to quickly experiment with ideas and get working MDD tools out to end-users quickly; it also aided in debugging and testing the tools. However, the performance of the tools obviously suffered, and when it came to processing larger models the impact of this decision was noticeable. It would be possible to build specialised versions of Epsilon that exploit code generation to meet strong requirements for higher performance. However, in previous experiments we have noticed that the bottleneck with (for example) execution time on model management operations tends to come from loading and storing models, not the virtual machines used to execute the MDD program. It may be advantageous to focus re-engineering effort on model persistence techniques first, before addressing code generators.
   Interpreters and code generators may co-exist, as demonstrated by the EMF-IncQuery framework, a follow-up project of VIATRA where model queries can be evaluated in both modes. While the advantages of the interpreted mode are similar to the Epsilon case, the compiled version significantly reduces integration efforts with existing tooling as only a few packages of generated Java code need to be integrated, instead of a complete query framework.

3.3 Modelling Technology

We learned several important issues related to the underlying *modelling technology* that MDD tools manipulate in the course of developing VIATRA and Epsilon.

1. *The choice of modelling technology impacts significantly on utility and flexibility.* VIATRA opted for a non-standard underlying modelling technology, called VPM, which is conceptually close to knowledge representation in the semantic web (like RDF documents and description logic). This design decision allowed us to easily implement rich transformation features such as generic transformations, multiple typing and dynamic re-typing of model elements, which were impossible (or extremely complicated) when using an EMF-based model representation and management. The downside of this is, unsurprisingly, that sophisticated bridges had to be built in VIATRA for EMF technologies, and significantly hindered industrial acceptance and usage. Epsilon, by contrast, focused on standard modelling technologies from the start, and

quickly developed an abstraction layer between MDD tasks and modelling technology.

2. *Ideally, the user of an MDD tool should not need to know how models are represented.* The tasks supported by an MDD tool – e.g., transformation, code generation, validation – are largely independent of how models are stored, and exposing end-users to modelling technology adds inessential complexity to MDD tasks. To this end, VIATRA is based on graph transformations and patterns, which abstract away from modelling technology. Similarly, Epsilon provides an extensible model connectivity layer that abstracts model representation from operations applied to models. This means, in part, that an operation in Epsilon is applicable to models of different types (e.g., EMF/Ecore, XML, MDR), and that new model representations can be added via implementation of new *drivers*.
3. *Providing access to model representation should be possible, but not required.* Sometimes – particularly, for improved performance – it is helpful to be able to access native model representations (e.g., XMI) directly, instead of having to work with them (abstractly) via drivers. This is particularly the case for large models, or for *new* types of models for which drivers or abstractions have yet to be developed or fully optimised. To paraphrase, it is helpful to allow end-users to work directly with the model representation if they need to (but, they shouldn't have to, since in some cases – particularly with XMI – working with native representations can be awkward, time-consuming and error prone). In certain tool integration scenarios, transformations may need to operate directly on native models developed with closed-technology tools, but this case should be avoided when developing *new* MDD tools due to immense impact on costs.
   More significant improvement on performance can be achieved by providing intelligent query techniques (like EMF-IncQuery). First, the native Java query API of EMF models provides poor performance in certain cases (e.g. lack of efficient reverse navigation, or type-specific enumeration of instances in a model). More importantly, the caching and incremental updating of query result sets provides substantially better performance in most cases compared to manually coded Java programs over the EMF query API.

3.4 Externalities

In this section we outline several lessons we learned about wider issues, particularly the relationship between the development of MDD tools and how they are used within larger system engineering processes, and by different kinds of users.

1. *Think about interoperation of the MDD tool with other tools early.* An MDD tool will often be used as part of a larger toolchain, in a larger system engineering process. For example, the results of model transformation may feed into downstream code generation, compilation, testing, quality assurance and dynamic update phases. Artefacts developed in such a process may all need to be stored under suitable version control. Early consideration of how the MDD tool will be used in concert with other tools in the lifecycle is likely to make future adaptation and deployment easier later. For example, in the case of Epsilon, a decision was made early on to support operation workflows using Ant, in order to support interoperation with existing tools (like compilers, test tools and version control tools) that were already supported via Ant tasks. In the case of VIATRA, transformation chains were driven by existing workflow engines (namely, the JBoss jBPM framework), and individual transformations were executed as services, which allowed combining automated and user-driven tasks into a single transformation process. Generally, standards – such as XML, workflow languages, metamodelling technologies – can help to support interoperation, though interoperation is not easy, particularly (as discussed earlier) if external APIs need to be used.
2. *Avoid over-automation.* MDD tools should only assist the developers in designing better systems, but they cannot substitute for them in making design decisions. Even the most sophisticated and optimized techniques will be ignored by the industry if they try to fully automate steps, which should actively involve the designer. So the developer makes design decisions and the clever MDD tool can sketch the options and automate the consequences.
3. *Consider the little things.* Users depend on documentation, built-in help, supporting infrastructure (e.g., syntax highlighting in editors, code completion) to complete their tasks. Reliability and significant features in the MDD tool are important, but additional features like documentation and developer productivity tools may be the difference between building a user-base and losing one.
4. *Rapid response to feedback can help you keep your users.* The availability of quick feedback to user questions on forums and newsgroups (e.g., the Eclipse forums) is not only a good way to build the user-base for your MDD tool, but is good advertising to users who are considering your tool.
5. *See what your user priorities are before spending significant effort on secondary features.* Secondary features like debuggers and code profilers are important, but are effort intensive to build and maintain. Monitoring user discussions in forums and feedback is essential to help identify where resources should be spent. For example, in the case of Epsilon, more

users were concerned with identifying performance bottlenecks than in carrying out detailed debugging of transformations or MDD operations; hence, effort was put into building profiling tools over a fine-grained debugging infrastructure. In the case of VIATRA2, end-user priorities introduced an oxymoron: the adjective "visual" only appears in its name (for historical reasons), but there was never a significant push from users' community to really introduce a visual language for capturing graph patterns and transformation rules as the textual language was sufficient for them.

4 Conclusions

Software engineers – and researchers in MDD in particular – have a particular mindset: they like to build new things, e.g., new applications, new tools to support system engineering. Many widely used applications and tools have evolved from a brilliant idea, or a brainstorming meeting, or a persistent engineer trying to solve a specific problem that had been troubling them. Many software engineers may be unable or unwilling to capture their insights and thinking during their work, and as a result some of their experience is lost. This paper is an attempt to capture some of our experience in developing and deploying MDD tools over the past ten years. Ideally some of the lessons we have learned will help in the development of the next generation of system engineering tools and languages still to come.

In looking back at the development of VIATRA and Epsilon, what seems in retrospect obvious is that while we have significant expertise in building MDD tools, there are still substantial open questions about how to use them in practice to solve large-scale software/systems engineering problems. In particular, how do we interface MDD tools with existing tried-and-tested engineering processes? How do we adapt the use of these tools to new processes and process standards? How do we validate the results of applying an MDD tool to a model? How can we qualify MDD tools with respect to relevant standards (e.g., for safety or security). There are substantial opportunities, and significant challenges, associated with understanding the interface with the wider context in which MDD tools are used.

What can we look forward to in the next ten years? Beyond the interface issue mentioned previously, some researchers are taking the view that MDD (and, indeed, software engineering research) is at a crossroads: that there are few fundamental scientific questions that remain to be answered, given the current state of software engineering practice. In our view, there are numerous open questions related to MDD *practice*, such as: how to best deploy tools; how to best use MDD tools with others; understanding the impact that MDD has on other engineering practices and processes. Using MDD tools – in anger, on real projects, with reported real results, is now both feasible and necessary. Now is the time to start focusing on *doing MDD*; the tools we have at our disposal are sufficiently mature to let this happen.

Acknowledgements. We thank Dimitris Kolovos, Louis Rose, Nikos Matragkas, James Williams, Fiona Polack, Antonio García Domínguez, Juan de Lara, Esther Guerra, Maider Azanza, Anne Etien, Davide di Ruscio, Alfonso Pierantonio, András Pataricza, István Ráth, Gábor Bergmann, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, and Gergely Varró. We acknowledge the support of the European Commission, the EPSRC, the Hungarian NDA and the Hungarian Academy of Sciences.

References

1. Acceleo. http://www.eclipse.org/acceleo/, 2011.
2. A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.
3. C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 807–810, New York, NY, USA, 2008. ACM.
4. A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró. Workflow-driven tool integration using model transformations. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science*, pages 224–248. Springer, 2010.
5. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pages 1280–1287, Dijon, France, April 2006. ACM Press.
6. G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *Proc. 4th International Conference on Graph Transformations, ICGT 2008*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2008.
7. G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. Efficient model transformations by combining pattern matching strategies. In R. F. Paige, editor, *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*, volume 5563 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2009.
8. G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. Incremental evaluation of model queries over EMF models: A tutorial on emf-incquery. In R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, editors, *Proc. ECMFA 2011: Modelling Foundations and Applications -*

7th European Conference, volume 6698 of *Lecture Notes in Computer Science*, pages 389–390. Springer, 2011.

9. G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *LNCS*, pages 76–90. Springer, 2010.
10. G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations: Change (in) the rule to rule the change. *Software and Systems Modeling*, 2011.
11. G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró. A graph query language for EMF models. In J. Cabot and E. Visser, editors, *Proc. International Conference on Model Transformation*, LNCS. Springer, 2011. Acceptance rate: 27%.
12. A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, 16(5):265–275, 2001.
13. S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level: the Fujaba approach. *Software Tools on Technology Transfer*, 6(3):203–218, 2004.
14. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In J. Richardson, W. Emmerich, and D. Wile, editors, *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pages 267–270, Edinburgh, UK, September 23–27 2002. IEEE Press.
15. J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom^3. *Software and System Modeling*, 3(3):194–209, 2004.
16. Epsilon. `http://www.eclipse.org/gmt/epsilon/`, 2010.
17. L. Gönczy, Z. Déri, and D. Varró. Model transformations for performability analysis of service configurations. In M. R. V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2008.
18. A. Hegedüs, A. Horváth, I. Ráth, and D. Varró. A model-driven framework for guided design space exploration. In *Proc. ASE 2011: 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, November 2011. In press.
19. W. Herzner, B. Huber, G. Csertán, and A. Balogh. The DECOS tool-chain: Model-based development of distibuted embedded safety-critical real-time systems. In *Proc. of the DECOS/ERCIM Workshop at SAFECOMP 2006*, pages 22–24. ERCIM, 2006.
20. A. Horváth and D. Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2011.
21. Á. Horváth, D. Varró, and T. Schoofs. Model-driven development of ARINC 653 configuration tables. In *29th IEEE & AIAA Digital Avionics System Conference (DASC)*, pages 5.A.5–1 – 5.A.5–115, Salt Lake City, US, 10/2010 2010. IEEE.
22. Á. Horváth, D. Varró, and G. Varró. Generic search plans for matching advanced graph patterns. *Electronic Communications of the EASST*, 6, 2007.
23. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *OOPSLA Companion*, pages 719–720, 2006.
24. D. S. Kolovos, R. F. Paige, and F. Polack. Merging models with the Epsilon Merging Language (EML). In *MoDELS*, pages 215–229, 2006.
25. D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proc. GaMMa*, pages 13–20. ACM, 2006.
26. D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.
27. D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Transformation Language. In *Proc. 1st International Conference on Model Transformation, ICMT*, Zurich, Switzerland, July 2008.
28. D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, pages 204–218, 2009.
29. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS*, pages 264–278, 2005.
30. R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*, pages 162–171, 2009.
31. I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Proc. First International Conference on the Theory and Practice of Model Transformations (ICMT 2008)*, volume 5063 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2008.
32. I. Ráth, A. Ökrös, and D. Varró. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software and Systems Modeling*, 9(4):453–471, 2010.
33. I. Ráth, D. Vago, and D. Varró. Design-time simulation of domain-specific models by incremental pattern matching. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008, Herrsching am Ammersee, Germany, 15-19 September 2008, Proceedings*, pages 219–222. IEEE, 2008.
34. L. M. Rose, D. S. Kolovos, N. Drivalos, J. R. Williams, R. F. Paige, F. A. C. Polack, and K. J. Fernandes. Concordance: A framework for managing model integrity. In *ECMFA*, pages 245–260, 2010.
35. L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model migration with Epsilon Flock. In *ICMT*, pages 184–198, 2010.

36. L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack. The Epsilon Generation Language. In *Proc. ECMDA*. LNCS, Springer-Verlag, 2008.
37. E. Schoitsch, E. Althammer, H. Eriksson, J. Vinter, L. Gönczy, A. Pataricza, and G. Csertán. Validation and certification of safety-critical embedded systems - the decos test bench. In J. Górski, editor, *Computer Safety, Reliability, and Security, 25th International Conference, SAFECOMP 2006, Gdansk, Poland, September 27-29, 2006, Proceedings*, volume 4166 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2006.
38. D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, October 2007.
39. D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and Systems Modeling*, 2(3):187–210, October 2003.
40. G. Varró, D. Varró, and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer, editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, volume 152 of *ENTCS*, pages 191–205. Elsevier, 2006.
41. VIATRA2 Framework. `http://www.eclipse.org/gmt/VIATRA2`.
42. xText 2.1. http://www.eclipse.org/Xtext/, 2011.