Open access • Journal Article • DOI:10.1007/S10270-012-0266-8

Bridging the chasm between MDE and the world of compilation — Source link

Jean-Marc Jézéquel, Benoit Combemale, Steven Derrien, Clément Guy ...+1 more authors

Institutions: University of Rennes, Colorado State University

Published on: 01 Oct 2012 - Software and Systems Modeling (Springer-Verlag)

Topics: Software system, Compiler and Program transformation

Bridging the Chasm Between MDE and the World of Compilation

Jean-Marc Jézéquel, Benoit Combemale, Steven Derrien, Clément Guy, Sanjay Rajopadhye

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

Bridging the Chasm Between MDE and the World of Compilation

Jean-Marc Jézéquel[1], Benoit Combemale[1], Steven Derrien[1], Clément Guy[1], Sanjay Rajopadhye[2] ⋆

[1] University of Rennes 1, IRISA, Inria
e-mail: {name.lastname}@irisa.fr
[2] Colorado State University
e-mail: Sanjay.Rajopadhye@colostate.edu

Received: date / Revised version: date

Abstract Modeling and transforming have always been the cornerstones of software system development, albeit often investigated by different research communities. Modeling addresses *how* information is represented and processed, while transformation cares about *what* the results of processing this information are. To address the growing complexity of software systems, Model-Driven Engineering (MDE) leverages Domain Specific Languages (DSL) to define abstract models of systems and automated methods to process them. Meanwhile, compiler technology mostly concentrates on advanced techniques and tools for program transformation. For this, it has developed complex analyses and transformations (from lexical and syntaxic to semantic analyses, down to platform-specific optimizations). These two communities appear today quite complementary and are starting to meet again in the Software Language Engineering (SLE) field. SLE addresses all the stages of a software language lifecycle, from its definition to its tooling. In this article, we show how SLE can lean on the expertise of both MDE and compiler research communities and how each community can bring its solutions to the other one. We then draw a picture of the current state of SLE, and of the challenges it has still to face.

1 Introduction

During the 70s, tools such as *Lex, Yacc, cc, Make* and *SCCS* basically defined the extent of software engineering tools. As time passed, these tools were extended, refined and produced offsprings in a variety of ways. Among the many research strands stemming out of this initial set, we will concentrate in this article on two that initially took very different routes, namely *Compilation* and *Model Driven Engineering* (MDE).

On the one hand, research in compilation significantly pushed the state of the art with respect to processing software language constructs. Major advances include efficient parsing and parser generation, advanced grammar formalisms (*e.g.* higher order attribute grammars [1]), source transformation systems and languages such as DMS [2], Rascal [3], Stratego [4], or TXL [5] that provide powerful general-purpose sets of capabilities for addressing a wide range of software analysis problems.Important contributions in program and dataflow analysis, include type checking, abstract interpretation, alias and shape analysis, and whole program analysis. On the code generation side, much progress has been made to address the variety and complexity of modern processors (*e.g.* complex instruction sets, power-consumption issues, hierarchical memory structures, multi-cores, etc.) with sophisticated algorithms handling (data and control) flow analysis, aliasing analysis, register allocation, etc.

A key driver of compiler research is the constant quest of maximal efficiency, both at code level and at meta-level (*i.e.*, in the algorithms implemented in the compilers themselves), to the point where we can truly speak of a culture of efficiency in the compiler community. However, a clear and recent concern is the extremely complex architectures of sophisticated compilers, that start to pose classical software engineering problems of reliability, development cost, maintainability, etc.

On the other hand, research in MDE originated in the problem context of representing and manipulating complex data. Beyond the initial issues of Chen's Entity-Relationship modeling, MDE was used widely enough to address Separation of Concerns (SoC) issues, by breaking down complex systems into as many models as needed in order to make all the relevant concerns understandable. These models may be expressed with a general-purpose modeling language such as the UML [6], or with Domain Specific Languages (DSL) when it is more appropriate. Each of these models can be seen as the abstraction of an aspect of reality for handling a given concern.

Models have long been used as *descriptive* artifacts, which was already extremely useful. The idea of MDE is to

⋆ This work is the result of a close collaboration between Inria and Colorado State University (CSU), involving two teams in MDE (the Triskell team at Inria and the SE group at CSU), and 2 teams in optimizing compilers (the CAIRN team at Inria and the Mélange group at CSU). This collaboration is partially funded by Inria associated teams MoCAA and LRS.

go beyond that, *i.e.*, to make it possible to perform computations on models, for example to simulate some behavior [7], or to generate code or tests from them [8]. This requires that models are no longer informal, and that the language used to describe them has a well defined syntax and semantics.

In many domains engineers rely on DSLs to solve the complex issues of engineering software at the right level of abstraction. These DSLs define modeling constructs that are tailored to the specific needs of a particular domain. When such a new DSL is needed, it is now often first defined through a meta-model, *i.e.*, a model describing its abstract syntax [9] when traditional language engineering would have started with the grammar of the language. Relying on well tooled standards such as MOF [10], the meta-modeling approach makes it possible to readily benefit from a set of tools such as reflexive editors, or XML serialization of models. More importantly, having such a tool-supported *de facto* standard for defining models and meta-models paves the way towards a rich ecosystem of interoperable tools working seamlessly with these models and meta-models, *e.g.*, Kermeta, which is a Model Driven Engineering platform for building rich development environments around meta-models using an aspect-oriented paradigm [7,11]. However, people building such tools often re-invent solutions that are well known in the compiler community.

The goal of this article is to explore how research in MDE and research in Compilation could cross-fertilize and might even converge on Software Language Engineering [12], after 30 years of diverging evolution. Several groups around the world have already started to investigate several aspects of this idea. For instance, Eclipse tools such as EMFtext[1] or Xtext[2] already bridge the world of meta-modeling and grammar parsing. JastAdd [13] combines higher order attribute grammars with object-orientation and simple aspect-orientation (static introductions) to provide better modularity mecanisms. Rather than reviewing such examples individually, we take a holistic view: what could compiler research bring to MDE (Section 2)? And conversely, what could MDE research bring to compilation (Section 3)? We show in Section 4 how the synergies between these two fields can benefit Software Language Engineering (SLE) as well as two key challenges which cannot be solved by these synergies. Finally, Section 5 concludes and presents some perspectives about other possible cross-fertilizations for SLE.

2 Leveraging Compilation Breakthroughs for MDE

Because computer programming evolved from machine language up through assembly and higher level languages, compilation has been one of the oldest areas of research in computer science. It has had a tremendous influence on and benefitted from many core areas in computer science.

In addition, modern compiler design practices also offer an interesting perspective from the software engineering point of view. It is indeed widely acknowledged that modern industrial strength compilers are extremely complex pieces of software, and that designing a compiler is a very challenging task. As a matter of fact, most modern compiler infrastructures are organized according to strict design rules, that are *best practices* and are the result of more than four decades of experience in compiler software design.

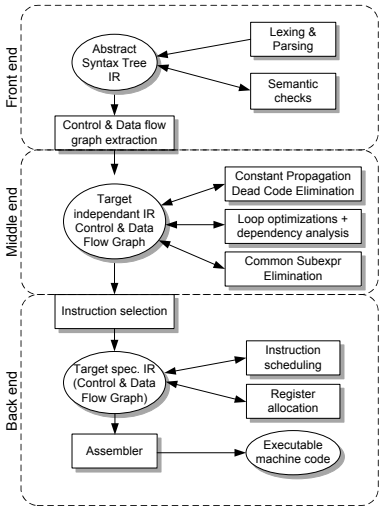

Figure 1: The typical organization of a compiler flow

This typical compiler toolchain organization is illustrated in Figure 1, and consists of three stages, namely the front-end which deals with parsing and semantic checks, the middle end stage, where all the machine independent analyses and optimizations are performed, and the back-end stage whose role is to efficiently generate efficient machine code.

We present in this section some experiences from compilation that could benefit MDE.

2.1 Parsers

One of the most obvious contributions of the compiler community to MDE lies in the design/proposal of efficient algorithms to automate the parsing of languages from formal specifications.

For context free grammars, the main breakthrough occurred more than 40 years ago with the introduction of LR and LL parser generators [14,15] that could efficiently deal with a large subset of context free grammars. Most programming languages are too complex to be expressed with such grammars, and these techniques were subsequently extended (*e.g.*, Generalized Left Right Parsing (GLR) [16] and LL(*) parsers [17]) to support a wider class of grammars.

Interestingly, compiler compiler frameworks started about 15 years ago to offer additional features which go well beyond strict parsing. As an example, smart error recovery schemes [18] are used either to help programmers find the cause of an error or to suggest possible corrections. Similarly incremental/lazy parsing [19] is also receiving more and more

[1] http://www.emftext.org/index.php/EMFText
[2] http://www.eclipse.org/Xtext

interest. This comes from the fact that syntactic analysis is nowadays tightly coupled to programming environment (*e.g.*, it enables on-the-fly syntactic checking). In this context parsing speed may become an issue, and hence requires adequate techniques and algorithms.

With the growing use of DSLs the MDE community faces a growing need for parsing tools, and has started to build on many achievements of the compiler research community. These achievements served as a base to build MDE rich language toolset. Indeed many languages have been introduced for a range of problems such as: GUI specification, verification and prototyping [20], multiagent system definition [21], model validation (*e.g.*, using the Object Constraint Language [22]), model-to-model transformations (*e.g.*, through dedicated languages such as the standard QVT [23] or ad hoc ones such as Kermeta[3] or ATL[4]) and model-to-text transformations (*e.g.*, template languages such as Acceleo, Xpand, etc). Note that all these language toolsets were designed in an ad hoc manner, with little reuse. This lack of reuse is one of the motivations behind SLE workbenches such as Xtext and EMFText.

2.2 Sophisticated algorithms

Most compilers perform very complex analyses or transformations on the program representation before targeting a given execution platform. These analyses rely on very sophisticated algorithms, that are generally either specific to the compiler target language or specific to a family of languages (imperative, with or without dynamic typing, functional, etc).

These algorithms (as explained in the next subsections) seek to obtain accurate information about the program behavior, while remaining scalable. They can be used for source code verification purposes (*e.g.*, for detecting memory errors at compile time), for optimization purposes (*e.g.*, finding statements that can be executed in parallel) or both.

Examples include *shape analysis* [24], *dependency analysis*, *pointer analysis* and *type inference* [25], just to name a few. Altough there exist many established algorithms for these problems, *static analysis* is a very important and very active facet/contribution of compiler research.

Similarly *loop transformations* are also key optimizations that focus on specific portion of a program and seek to adapt the ordering of computations in loop nests kernels to suit the target machine memory hierarchy (cache memory, processor registers) or machine parallel processing capabilities (thread or data level parallelism). Of course *machine dependant* algorithms have also received a lot of attention. Such algorithms include register allocation and instruction selection and instruction scheduling algorithms which seek to find the best mapping between a program and a given machine instruction set or micro-architecture.

Again, such a transformation framework leverages very complex algorithms, involving many challenging combinatorial optimizations problems and platform-specific knowledge.

On the other hand, because the idea behind MDE is to provide models in which most domain specific knowledge is made explicit in the representation, developers do not generally need (or do not seek) to rely on sophisticated tools and algorithms to derive their systems implementations. Indeed, even though the implementation of many model transformations consists of very complex structural transformations, their complexity derives from the complexity of the models involved in the transformation, rather than from the algorithm used in the transformation. However, since modern MDE increasingly advocates the use of DSLs to describe software systems, compilation chains for these DSLs will eventually have to consider more complex transformations, even with the use of more suitable structures for model transformations. Therefore, MDE could benefit from the long time experience of the compilation community in the development of such complex algorithms.

2.3 Program transformation paradigms

Most compiler representations are either based on (i) a tree based structure, which more-or-less directly corresponds to the Abstract Syntax Tree (AST) of the target language, or (ii) on a Control Flow Graph (CFG) representation in which the high level control structure is flattened to expose the program as a graph of basic blocs in which the execution is strictly sequential. Usually, most compiler infrastructures manipulate both representations (front end analyses and transformations operate on ASTs whereas middle and back end stages generally operate on CFGs).

Many transformations/analyses operating on AST (resp. CFGs) have a lot in common. Therefore, there has long been interest in proposing program transformation/analysis paradigms to enable capturing and expressing a large family of transformations within a single workbench [26,27].

Transformations known as *term rewriting techniques* [28, 4] fall into this category as they allow compiler writers to concisely express complex pattern matching and rewriting operations on trees. Term rewriting techniques make it possible to combine powerful pattern matching techniques with high-level rewriting 'strategies' that allow users to define the tree term traversal in a concise and yet very flexible way, by combining simple low-level primitives. In addition to conciseness and expressivity, the underlying formal definition of such term rewriting techniques also brings additional opportunities for formal proof and verification (*e.g.*, completeness, consistency). MDE tries leverage such techniques for model transformation. Among others we can cite tools such as AGG[5] or TOM-EMF[6]. This is also the goal of the QVT [23] standard which defines a declarative/operational hybrid language for

[3] http://www.kermeta.org/
[4] http://eclipse.org/atl/
[5] http://user.cs.tu-berlin.de/~gragra/agg/
[6] http://tom.loria.fr/wiki/index.php5/Documentation:EMF

model transformations, although the standard does not give any reference implementation for this language.

2.4 Program analysis paradigms

Another class of popular compiler analyses/transformations, which are usually applied to CFG like structures, are those based on *dataflow analysis* [29]. These analyses are used to gather information about the possible values manipulated at different control flow points of a program, the information collected then being used to drive verification and optimization processes.

The analysis is based on the principle of *dataflow equations*, which are used to propagate information between pairs of connected basic blocks (either *forward* along or *backward* against the actual program control flow) using a *transfer function*. These equations are then solved iteratively until a fixed point is obtained.

Because of the genericity of the technique, several dataflow analysis workbenches (some of them leveraging DSLs) have been proposed [30,31,32]. Their goal is to help automate part (or all) the implementation of a dataflow analysis for a given compiler infrastructure and a (family of) languages.

At first glance, it may seem that these advanced techniques are very specific to general-purpose programming languages from which a control flow can be extracted[7]. However, many domain-specific languages embed constructs to describe behaviors, these behaviors following either imperative or functional programming semantics. All these languages are hence also amenable to dataflow analysis, and can therefore benefit from all existing dataflow-based Verification and Validation (V&V) techniques.

2.5 Efficiency, scalability

Compiler designers always emphasize scalability when integrating or devising new analyses and transformations in their flow. As an illustration, production compilers such as gcc[8] or Intel's icc[9] are expected to compile a 100k-loc program in a matter of minutes, whether or not optimizations are enabled. These constraints are even tighter for just-in-time compilers, which are becoming more and more widely used due to the wide adoption of virtual machines such as the JVM and .Net. In these latter cases, compilation time directly impacts program execution and only very fast and scalable algorithms (*i.e.*, linear complexity algorithms) can be allowed.

Achieving such a scalability while preserving good performance in the compiled code is therefore a very difficult challenge, especially since many of the optimization problems involved in compiler back-ends (register allocation, instruction scheduling and selection and other combinatorial optimization problems) are NP-complete or NP-hard and suffer from combinatorial explosion.

In the MDE world, performance has not been such a concern so far. The software engineering community has constantly been benefiting from the Moore's law driven progress of Very-Large-Scale Integration technology combined with the numerous breakthrough in programmable processor architectures (RISC, super-scalar ISA, etc.). Such evolutions enable the introduction of more and more complex frameworks, whose role is to abstract away the underlying machine as much as possible. As a consequence, and from an optimizing compiler point of view, modern software engineering workbenches are perceived as resource hungry, slow and suffering from significant scalability issues.

As a matter of fact, when it comes to execution efficiency, the MDE community has been long confronted with a chicken-and-egg dilemma. Performance issues will only be addressed by the community when there is a sufficiently large user base with such strong performance requirements. In the mean time, potential users seeking good performance will remain reluctant to use MDE technologies. We believe this dilemna will soon have to be addressed by the MDE community. Indeed, the outbreak of multicore and manycore architectures will require a major shift in the way software is designed, since benefiting from processor performance improvements will no longer happen for free. Among other challenges, software implementations will be forced to explicitly expose considerable amounts of *usable* parallelism and this could be the first step toward a more systematic concern for implementation efficiency. Consideration of such efficiency and scalability issues would benefit MDE, and particularly in domains such as embedded systems, green computing or high performance computing.

2.6 Platform Description Model

Compilers are meant to produce efficient code, and producing efficient code obviously requires a deep knowledge and understanding of the target execution platform. For this reason, there has been a lot of effort on *optimization*, *i.e.*, to improve the efficiency of the code generated by compilers.

The problem tackled in the optimizing compiler community goes from high level target agnostic optimization (dead code elimination, constant propagation, common subexpression elimination, etc.) to more machine dependent ones (register allocation, instruction selection, automatic parallelization, etc).

Because of the high development cost required to port an existing compiler to a new processor architecture, significant research was devoted to trying to automate part of this effort, by making the infrastructure retargetable [33].

In optimizing compilers, target description languages are commonly used to describe in a systematic manner, the fea-

[7] This includes functional languages where the notion of dataflow analysis is well known

[8] `http://gcc.gnu.org/`

[9] `http://software.intel.com/en-us/articles/intel-compilers`

tures of interest of the underlying platform. In addition to target portability they expose some cost feature to the compiler/code generator so that the quality of the generated code can be optimized. Such concerns are usually distant from the MDE community.

This is made possible through the use of a formal machine description, which captures all the information/knowledge about the target platform [34,35]. As an example, most compiler infrastructures are designed to be easily retargeted to a new processor. This retargetability is achieved through the use of a formal description of the processor instruction set (operational semantics, execution cost, binary format, etc). This description is then used to automatically regenerate an optimizing back-end for the new architecture (code selector, register allocation, assembler, etc).

Later, a similar principle was also followed by the MDE community: for example Model-Driven Architecture (MDA) [36] aims at achieving a good separation of concerns through the use of Platform Independent Models (PIM) and Platform Specific Models (PSM). The transformation from PIM models to PSM models is then performed using a Platform Description Model (PDM) whose role is to provide an abstract description of the target platform that is supposed to drive the transformation process [37,38].

Experience shows that only few MDA workbenches actually follow this approach. In practice most PIM to PSM transformations do not use an explicit model of the target platform: the platform specificities are instead implicitly captured within the transformations. Even when PDM are explicitly used in the flow, they almost never serve to optimize the performance, or other QoS aspects, of the generated code, unlike what a compiler would do.

Their weakness is that most PDMs do not expose enough semantics and/or details to enable optimized PIM to PSM transformations. This lack of formalization of platform features can be explained by the difficulty of describing complex software platform (such as J2EE, .NET, Android, etc.) at the right level of abstraction. As far as this issue is concerned, the MDE community has probably a lot to learn from the optimizing compiler community.

3 Leveraging MDE Breakthroughs for Compilation

Model driven engineering (MDE) is the result of a long evolution in software engineering to handle the increasing complexity of software development. In particular, MDE benefits from the modeling and programming evolution and their best practices such as complex data representation, separation of concerns or design-by-contract.

As shown in Figure 2, MDE workbenches often use high-level models to describe a system, each one representing an *aspect* of the system. These models are then composed and transformed into lower level models in successive steps until we obtain an executable representation of (some part of) the system (*e.g.*, executable code, configuration scripts, etc.). Moreover, this process can also include early Verification and Validation (V&V) stages at every step of the refinement, by the transformation of the (domain specific) models into suitable models for V&V (*e.g.*, petri nets for model checking).

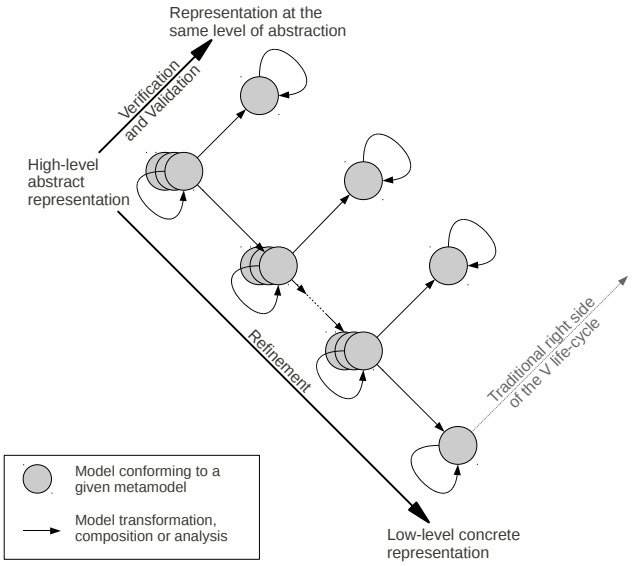

Figure 2: MDE in the left-hand side of the V life-cycle

This process, which is represented by the refinement axis, is of course very similar to compilation which produces a low-level executable code from a high-level source code by successive semantic-preserving steps of transformations. We now present some of the best practices of MDE in software development and describe how compilation could benefit from them.

3.1 Complex Data Representation

MDE is primarily concerned with reducing the accidental complexities associated with developing complex software systems [39]. This is accomplished through the use of technologies that support rigorous analysis and transformation of abstract descriptions of software to concrete implementations [40]. To handle the design of such descriptions, MDE technologies take advantage of the experience acquired in the fields of databases and programming languages, and thus integrate more advanced paradigms to represent complex data.

In particular, the object-oriented principles are shifted from programming to modeling with languages such as UML [6]. Nowadays, these principles are at the heart of the domain-specific knowledge capitalization in the metamodeling activity thanks to languages such as MOF [10]. Consequently, a metamodel can be seen as the object-oriented model of an Abstract Syntax Tree (AST) and the corresponding symbol table [41,9].

Usually, metamodeling environments also provide built-in support for serialization, model interchange, visualization and more recently configuration management. Thus, one obvious way that MDE can contribute to the world of compi-

lation lies in the best-practices for representing and manipulating the complex data structures at the heart of modern optimizing compilers.

Moreover, the high-level information contained by models can be used for domain-specific optimizations, whereas the same information may be difficult for a compiler to extract from a lower-level representation (*e.g.*, a classical compiler IR) [42,43].

3.2 Separation of Concerns

Each model represents an *aspect* of the software (aka. viewpoint) and allows a clear *separation of concerns* (SoC) in the development activity. Thus, each domain expert can focus on her very particular problem and benefit from a specialized toolset without having to consider any other concerns. This principle is illustrated in Figure 2 where different models, conforming to different languages supporting dedicated tools are used to represent a software system. These models can then be composed and refined into models at a lower abstraction level which capture new concerns (as shown on the refinement axis on Figure 2). Moreover, at each step of the refinement, it is possible to transform the models into suitable models for an early V&V of the system (as shown on the V&V axis of Figure 2). The V&V concern is thus separated from other concerns, thanks to the use of specific models.

SoC could benefit the compilation community to enable a clear identification of the different concerns in a compiler toolchain (performance optimization, semantic analysis, translation between languages, etc.)

Each concern in a compilation chain could be addressed through dedicated models that would then ease the design of the compiler passes. Such SoC is already (partially) achieved in most compilation workbenches, which generally rely on several different intermediate representations (IRs) corresponding to the different stages of a compiler toolchain. These IRs range from enriched ASTs that are used during the early stage of the compilers, to low level machine specific descriptions of the program, which serve for the back-end stage. Nevertheless, each IR of a compiler mostly corresponds to a complete/full featured description of the program, and almost always contains information that is irrelevant to the transformation (or analysis) at hand. For example, register allocation is performed on an IR which contains a complete machine level description of the program, even though the algorithm only needs an enriched conflict graph structure to operate.

3.3 A Uniform Model-Driven Approach for Software and Language Engineering

One of the main contribution of MDE is to provide methodological and technological foundations to design and implement DSLs and their tooling. For this purpose, MDE leverages a similar approach as the design and implementation of systems (object-oriented paradigm, design by contract, etc). This approach makes the modeling of a DSL (aka. metamodeling) very similar to the modeling of a system. Indeed, modeling a software system or a modeling language for software systems end up being the same, and can therefore be tackled within a unified workbench.

While the abstraction of a software system is called a model, the model of the abstract syntax of a language is generally referred to as a metamodel. The modeling of a language is enabled by a metamodeling language or metalanguage (which is to metamodels what a modeling language is to models) such as MOF. It is then possible to define tools on this metalanguage (*i.e.*, metatools), and particularly generative metatools. This enables the tooling of metamodels conforming to the metalanguage, by automating all or part of the development tasks (*e.g.*, textual and graphical editor generator, simulator generator, etc). Structural software design patterns are a perfect example of such generic concepts that can be shared between metamodels. As a consequence, expressing them at the metalanguage level provides a powerful toolbox to the developers who can apply or reuse these patterns into all their metamodels.

The notion of generative metatools is not new to compilation community. For example, BEG [44] and BURG [45] are two tools which are used to generate processor instruction selectors, and PAG [30] and DFAGen [32] are two dataflow analysis generators. Such tools could benefit from a shift from the program level (*i.e.*, model level) to the programming language level (*i.e.*, metamodel level) of compilation techniques (*e.g.*, transformation paradigms presented in Section 2.3).

Moreover, instances of IRs are abstractions of the compiled program, and thus are models. Therefore, the use of metamodels to define the abstract syntax of the intermediate languages seems a natural choice. In this context, each pass of the compilation chain becomes a model transformation.

Two examples of compiler infrastructures based on metamodeling and model transformations are GeCoS[10] and AlphaZ[11]. These two research-oriented optimizing compiler infrastructures have faced the same software engineering issues during their development, and decided to use MDE to tackle them [46]. Indeed, compilers are composed of multiple passes manipulating several intermediate languages. As such, compiler developers face well known software engineering challenges such as maintainability of the code, documentation production or time consuming and error prone development tasks. With the use of metamodeling comes a homogenization of development practices such as naming conventions. Metamodels also offer an abstract representation of the software, and document many important design choices. These abstract representations include an object-oriented graph (*e.g.*, with the notion of specialization/generalization) and a tree (the containment tree). Moreover, they offer through a graphical representation an holistic and structured view of the software. Additionally, metatools and meta-

[10] `http://gecos.gforge.inria.fr`
[11] `http://www.cs.colostate.edu/AlphaZ/`

tooling greatly help in automating many of the time consuming and error prone development tasks. Finally, we observed that metatools and generative approaches operate as creativity boosters as they enable very fast prototyping and evaluation of many new ideas.

3.4 Design-by-Contract

Design-by-contract [47] has been first proposed in object-oriented languages as a way to express assume-guarantee conditions [48] on the behavior of software by edicting precise invariants and pre and post-conditions on its execution. It is now integral part of MDE through languages such as the Object Constraint Language (OCL) [22], allowing to use assume-guarantee conditions on models and model transformations as for any other software. Thus, it is possible to express conditions on the input and the output of a transformation, or of a compilation pass. These conditions can be used to ensure a sound combination of the successive passes, driving the design space exploration with respect to the current state of the compiled program and the desired result. Preconditions can express the expected state of the input representation of the program and post-conditions can express the result in terms of optimization metrics (*e.g.*, performance cost models, code/memory size, parallelism). Such information are generally only known by the developers of compiler analyses and transformations and implicitly expressed by the order of the compilation passes, making it difficult to design a modular compiler.

Table 1: Solutions from compilation to MDE shortcomings

MDE shortcomings	Compilation solutions
Increasing need for parsing tools due to increase in number of DSLs	Efficient parsing and parser generators
Platform Description Model	Capture of platform specific knowledge through dedicated descriptions
Tool efficiency and scalability	Sophisticated algorithms and heuristics
Increasingly complex model transformations	Know-how in sophisticated algorithms development and program transformation paradigms

Table 2: Solutions from MDE to compilation shortcomings

Compilation shortcomings	MDE solutions
IRs contain more and more complex information, more and more complex IR processings,	Complex data representation and Separation of Concerns
Maintainability	Homogeneization of software through generative approaches
Documentation	Metamodels as documentation
Error-prone and time consuming development tasks	Automation through metatools and metatooling
Ordering of the compilation pass	Design-by-Contract to limit possible choices to meaningful choices

4 Convergence into SLE

As seen in the two previous sections, the compiler research community has already proposed some solutions relating to several of the problems faced by MDE (*e.g.*, efficient parsing, platform specific knowledge capture, scalability and efficiency issues). This is also true the other way round (*e.g.*, complex data representation, separation of concerns and design by contract). These solutions to shortcomings from the two communities are summarized in Tables 1 and 2. These tables reflect the focus of each community: compilation emphasizes *what* is the result of data transformation (to a first approximation, more efficient code) whereas MDE concentrates on *how* data is represented (well defined domains able to better capture specific knowledge and know-how).

Cross-fertilization of these two worlds hence leads to an engineering of software languages that addresses both the representation of data (*i.e.*, the design of tool-supported software languages) and the analysis and transformation of this data (*i.e.*, the implementation of supporting tools for such languages). Some recent work has already been following this direction, by providing generic tools for language design and implementation.

In this section we present the road already covered in the cross-fertilization of the compilation and MDE worlds (Subsection 4.1) as well as the road we still have to cover (Subsection 4.2). We point out two challenges we believe to be of high interest to Software Language Engineering (SLE): the increasing number of software languages, and the need to bring V&V methods into SLE.

4.1 The Road Already Covered in Cross-fertilizing

The term *software language* refers to all the kinds of artificial languages which are implied in software systems development, including programming and modeling languages but also data models, DSLs or ontologies [49].

The number of such languages is constantly increasing [12, chap.1], mainly due to two reasons. The first one is the increasingly broad spectrum of domains addressed by software systems (*e.g.*, avionics, home automation, etc.), raising the need for languages to care for the specificities of these domains, along with a need to make language design and implementation methods accessible to non-computer scientists (*i.e.*, domain experts).

The second reason is the ever growing size and complexity of software systems, leading to a need for breaking down the systems into smaller understandable pieces (objects, aspects, etc).

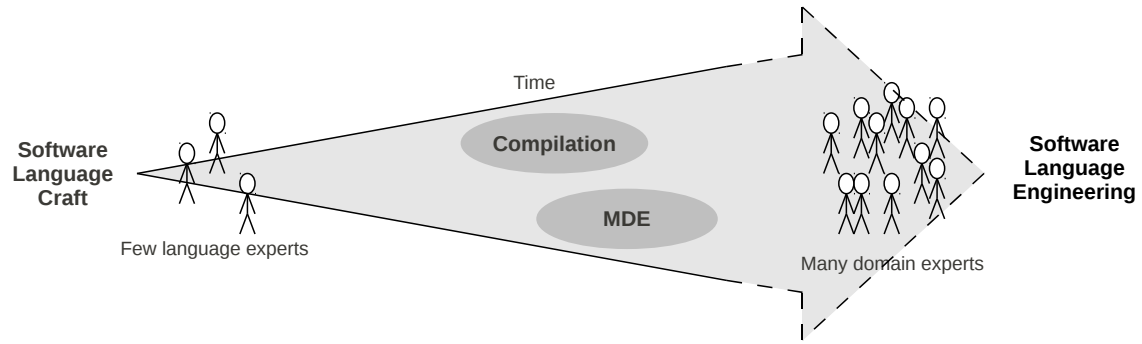

Figure 3: The role of MDE and compilation communities in the evolution from a craft to an engineering of software languages; opening the way to design and implementation of software languages by few language experts to many domain experts

These two reasons lead to the continuous creation of new languages, as well as the evolution of old ones [12].

This increasing number of software languages hence brings a need to shift from a software language "craft", practiced by only a few language experts, to a systematic approach for the design and implementation of tool-supported software languages, that is usable by a large number of non-expert users. There is a need to create new tools and methods for the design and implementation of languages as well as a need to bring existing tools to domain experts. Some of these tools and methods have existed for a long time. Such tools include the BNF and EBNF metalanguages, and lexer and parser generators like *Lex* and *yacc* allow the automatic tooling of the languages defined from these metalanguages. However, mastering even such well established tools requires an amount of work that is generally not affordable to domain experts. Moreover, defining an entirely new language for every system concern is not an option when considering the effort needed. One of the goals of SLE is to provide accessible tools and methods addressing all the stages of the software language lifecycles, from design and implementation to use and evolution.

MDE fosters the use of DSLs for the representation of software systems and provides theoretical and methodological foundations for the design of these DSLs. Moreover, MDE is interested in manipulating models (*i.e.*, *description artifacts* coming from modeling languages). Meanwhile compilation develops tools for languages of different programming paradigms (imperative, functional, etc.) and compilers manipulate programs (*i.e.*, *description artifacts* coming from programming languages) through different transformations and analyses.

Because SLE is interested in all the software languages, and therefore all the *description artifacts* coming from these languages (*e.g.,* models, programs, etc.; also called *mograms* by Kleppe [12, chap. 3]), the cross-fertilization of MDE and compilation can bring the experience of both communities to SLE. Compiler and editor generators already go into this direction and allow domain experts to generate tools for their languages based on language specifications. Tools like XText and EMFText provide concrete textual syntax relying on the abstract syntax of the language, and generate tools such as parsers, lexers, serializers or editors with syntax highlighting, on-the-fly code completion and correction. To do so, such tools rely on a metamodel describing the abstract syntax of the language and, for example, on the LL(*) ANTLR parser generator workbench, so as to offer understandable representation of the abstract syntax and fast, flexible parsing. The TOPCASED project[13], an open source MDE toolkit for the design of safety critical applications and systems [50], goes even further providing a graphical editor generator as well as simulator generator and facilities to use several V&V techniques (*e.g.*, model checking).

Figure 3 illustrates how MDE and compilation contribute to the design and implementation of tool-supported software languages, putting them within the reach of domain experts (*i.e.*, people who are not software language experts, even not computer scientists).

4.2 Still a Long Way to Go in Cross-fertilization

There still remain many challenges ahead for SLE to provide a complete workbench for the whole lifecycle of software languages, if this workbench aims for broad adoption by domain experts. Domain experts need to manipulate concepts relative to their domain only. This could be achieved by automating processes out of the given domain and offering facilities to design and implement DSLs and their respective tooling.

Some of these facilities already exist (*e.g.*, partial automation and facilities for the language design and implementation) but we have still to cope with the increasing number of software languages.

Language designers also need to make sure that the tools they use enforce some properties and that they do not seriously modify the nature of their work (*e.g.*, the binary code

[12] Note that DSLs are also a way for companies to protect their Intellectual Property Rights on their knowledge capitalization.

[13] Toolkit in OPen-source for Critical Applications & SystEms Development, cf. `http://www.topcased.org`

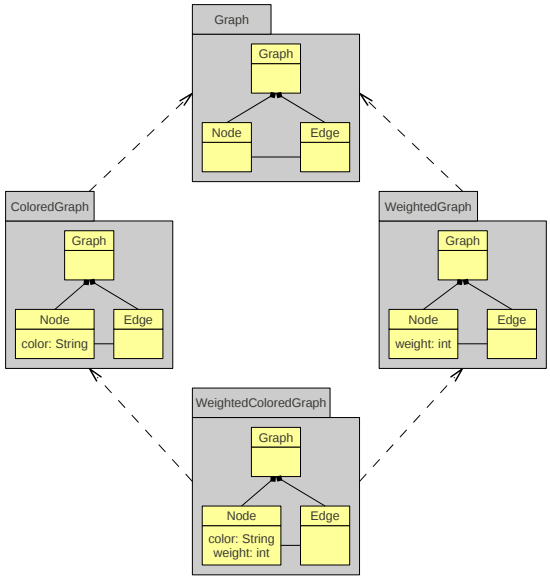

Figure 4: A family of graph languages

produced by a compiler must preserve the semantics of the source code).

Some of these needs have been addressed in different contexts by other fields of computer science, particularly in programming languages semantics and theory, and formal methods. The expertise developed in these domains must also to be incorporated into SLE, along with many other results from other fields and/or communities of computer science.

4.2.1 Amortization of Language and Tool Definition and Implementation As mentioned previously, the number of software languages defined and used is growing. This increase is there to answer the various needs of an equally growing number of domain experts. However, methods enabling the construction of a language from other existing languages, and for the reuse of different tools (*e.g.*, analyses and transformations) between such languages, are still lacking.

MDE provides tools to ease the design of software languages. However, these languages are most often still created from scratch even though they could benefit from concepts and structures defined in already existing languages. An engineering of software languages hence needs to provide methods to enable the derivation of a language from another one, but also to incrementally define families of languages.

Figure 4 presents an example of such an incrementally defined family of languages, where dashed lines represent the derivation from a language to another. This family is composed of graphs, colored graphs, weighted graphs and weighted colored graphs, where each language shares some concepts with the others (namely *Graph*, *Node* and *Edge*) while specializing them (*e.g.*, adding attributes such as *color*). This situation leads to a lattice of graph languages inheriting from one another.

Moreover, manipulations (*e.g.*, analyses and transformations) written for the new languages are most often implemented from scratch whereas at least a part of them is already implemented for several other languages and could therefore be reused. Here again, such facilities should be integrated in the engineering of software languages.

For example, Dead Code Elimination (DCE) is a classical optimization in a vast majority of compilers for imperative languages. However, a similar optimization can be done on a hardware circuit description to eliminate useless hardware components (*e.g.*, components which are not connected to an output), or in functional languages. The basis of DCE is a reachability analysis processed on a Control-Flow Graph (CFG), all unreachable code blocks then being removed. In a hardware description, we can do a similar reachability analysis, using output ports as roots, to find and remove useless components. Rather than implementing such an analysis for each language, it would be interesting to amortize the effort between all the languages with a minimum of adaptations.

Figure 5(a) illustrates one way to cope with this issue, where plain arrows represent *mogram* manipulations such as analyses or endogenous transformations (*e.g.*, T1 or T2) or exogenous transformations (*e.g.*, L1 to L4). Here, a total of six *ad hoc* transformations are needed to use the two analyses defined on languages L4 and L5 on *mograms* coming from L1, L2 and L3. Many *mogram* manipulations (six transformations and two analyses) are developped as the two analyses have been implemented separately for each language.

To solve this problem, pivot languages (cf. Fig. 5(b)), which are intermediate languages between two sets of languages, have been intensively investigated in the last decade, *e.g.*, in V&V and for domain-specific model checking purposes [51,52,53,54]. Their interests are twofold:

- decrease the semantic gap between two sets of languages to ease the design of translations between them,
- capitalize and share some translation passes.

Nevertheless, the mere definition of the concept of pivot language raises several problems. In particular, the concept of pivot language implies a certain universality of the pivot. Every concept possibly defined in existing or future languages has to be somehow included in the pivot. The existing work has shown the difficulty of defining such a universal "union". For example, TOPCASED has defined the language Fiacre [54], inspired from V-Cotre [52] and NTIF [51], which is a pivot language between the DSL of the IDE (*e.g.*, UML, AADL, etc.) and the various V&V environments. This language has reduced the distance between the DSL semantics and the formalisms dedicated to the V&V (namely, Petri nets and timed automata), and allowed each DSL to share the transformations from the pivot language to dedicated V&V formalisms. In practice in TOPCASED, the design of a pivot language allowing the translation of very different languages requires very different concepts and appeared to be quite impossible. This difficulty has led to design a pivot language family, such as synchronous and asynchronous versions whose combination also proved to be difficult. Even if

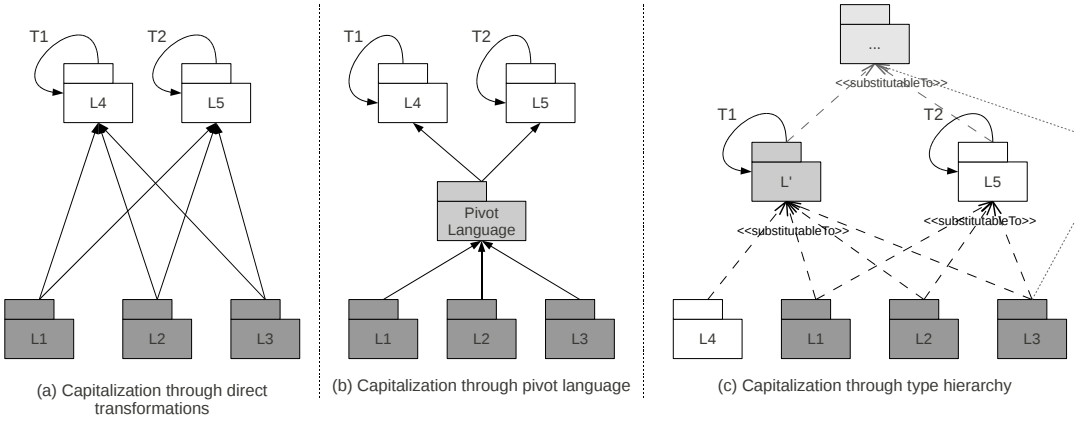

Figure 5: Some approaches for the capitalization of model manipulations

this approach is interesting for the reuse of some model manipulations, the concept of pivot language is hence difficult to implement in the general case (*i.e.*, for any set of languages).

Several other approaches have been proposed within MDE in order to enable the reuse of model manipulations over a family of metamodels [55]. One of the issues faced by these approaches is the presence of structural heterogeneities between the metamodels of a family. As an example, consider two graph metamodels, the first one where edges are modeled with a class *Edge* and the second where edges are modeled as references from class *Node* to itself. A manipulation implementing a graph traversal defined on one of these two metamodels cannot easily be reused on the other, although they clearly belong to the same family. To cope with this problem, the existing approaches generally use one or two of the following mechanisms:

- adapting the metamodel on which the model manipulation will be reused (*e.g.*, adding a class *Edge* in the second metamodel);
- adapting the manipulation to be reused (*e.g.*, automatically generating a new manipulation which traverses a graph).

Some examples of approaches using the first solution are those which introduce genericity at the metamodel level, by means of *templates* [56], as well as *concepts* and *mixin layers* [57]. Other approaches which adapt the metamodel include patterns with variable entities (*i.e.*, patterns expressing the needed concepts only) for declarative model manipulation rules [58], bidirectional model manipulation DSLs [59] and static introduction in order to make two metamodels structurally equivalent [60]. Wimmer *et al.* presented an hybrid approach (*i.e.*, adaptation of the metamodel and the model manipulation) based on above mentioned *concepts* that generates a metamodel-specific manipulation from a generic one defined for a metamodel family [61].

Incremental design and manipulation reuse have also been largely addressed in programming languages, especially in the object-oriented paradigm. Facilities such as inheritance, genericity, subtyping and polymorphism allow two kinds of reuse through:

- the incremental design and specialization of types and classes, and thus, the reuse of structure (signature) and implementation,
- the implementation of manipulations for a family of types rather than for each members of the family.

To bring some of these facilities into SLE, languages should be considered as first-class entities, *i.e.*, types, hence enabling relationships such as inheritance or subtyping between languages. Since MDE represents the abstract syntax of a language as a metamodel (*i.e.*, a set of classes and their relations), this has been made possible by leveraging work on type groups [62] and family polymorphism [63] which allow such relationships between families of classes. The introduction of inheritance or subtyping relationships then enables reuse across languages and reuse of *mogram* manipulations.

Model typing is an existing approach introduced in MDE as a way to allow reuse of model manipulations and incremental design and implementation of modeling languages through typing relations. In this context, Steel *et al.* introduced the notion of *model type* as the set of object types for all the objects contained in a model and their relations [64]. Based of this notion of model type, we have defined four subtyping relations between model types [65]. These relations can be used to implement a model-oriented type system providing facilities such as reuse of model manipulation and incremental language definition (by means of incremental model type definition), but also auto-completion, impact analyses or type-guided compiler optimizations.

We believe that such a type system for *mograms*, including *mograms* and *mogram* manipulation typing, inheritance, genericity and subtyping would enable new possibilities for the design and implementation of tool-supported languages. Indeed a *mogram* type including both object types for objects which could belong to a *mogram* and signatures of manipulations (such as introduced by Vignaga *et al.* [66]) defined on this *mogram* would contain abstract syntax (a set of object types can be seen as a metamodel) and semantics given

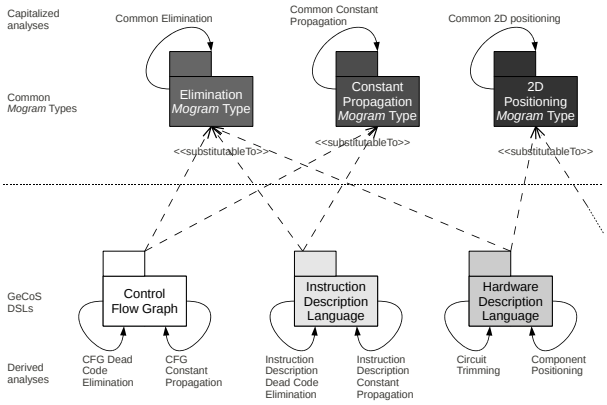

Figure 6: Capitalization of model manipulations in the GeCoS research-oriented compiler infrastructure using model typing

by the manipulations (*e.g.*, translational semantics given by a code generator) of a language. In this context, inheritance between languages (or *mogram* types) would enable incremental design of languages through inheritance and specialization of structures, and genericity and subtyping would enable the reuse of *mogram* manipulations by introducing substitutability of *mograms*.

One advantage of such facilities compared to pivot languages is the possibility of defining a type lattice, where a type is the "intersection" of its subtypes and not the "union" as in a pivot language. This way a language can inherit from the manipulations of interest defined at the highest possible level, *i.e.*, the level in the type hierarchy which contains the minimal set of necessary information to express the manipulations. This is illustrated in Fig. 5(c) where dashed arrows represent the subtyping relationships between two languages and thus the substitutability of the *mograms* coming from the "sub-language" to the *mograms* coming from the "super-language" in manipulations (*e.g.*, *mograms* from L2 and L3 can be used *as mograms* from L5 for the manipulations defined on the latter). To follow this approach, it is however mandatory to precisely define how a language and a *mogram* type are related, and then how such a type can inherit from another one.

As an example, the various DSLs used by the GeCoS compiler infrastructure: CFG, hardware description language (HDL) or processor instruction description language are illustrated in Figure 6. It is possible to perform several analyses on *mograms* coming from these DSLs. For example, analyses which eliminate useless parts of the *mograms* such as dead code elimination or circuit trimming, but also more specific analyses such as loop optimizations on CFG. Several analyses, while performed on *mograms* coming from different formalisms (data or control flow) present similarities and could be written only once for a common *mogram type* and shared between the DSLs by subtyping or genericity instead of being written again and again.

4.2.2 Inter-language Reasonning With the growing size of software systems, and particularly with the growing number of languages used for the design and implementation of one system, there is an increasing need for globally reasoning on a system through different viewpoints. To reason about such a system, a tool would need to be able to browse the links (implicit or explicit) between the languages in which viewpoints are defined.

For example, checking the consistency of several views of a system requires knowing which elements of these views are related (*i.e.*, which elements have to be consistent with each other) and how they are related (*i.e.*, how to check the consistency of a set of related elements). The links indicating which elements are related can be implicit (*e.g.*, elements with the same names are related) or explicit.

DeRemer and Kron [67]; working in the context of programming-in-the-large vs. programming-in-the-small, were probably the first to introduce such explicit links. They introduce the notion of modules written in a language and linked by another language, dedicated to this goal.

Of course, ever since the dawn of the history of software systems, compilers have created such links between languages, mapping structures from high-level languages to their equivalent in processor instructions sets, but these links were generally implicit and embedded into the transformation from one language to another.

The possibility of defining relationships between several languages, in order to reason about these relationships, has been already explored in MDE. For instance, Megamodels are models whose elements are themselves models linked by various kinds of relationships (*e.g.*, conformance, transformation, etc.) [68]. They were primarly meant to manage model dependencies [69]. Other approaches use relationships between languages to ensure the consistency between models of the same system [70], to help the design of complex systems [71], to specify transformations from one language to another [72,73] or to ensure the traceability of such transformations [74].

We believe that making these links explicit and clearly separating them from the semantics they carry should ease their manipulation as well as their reuse. Hence such structure should be an inherent part of a metalanguage.

4.2.3 Verification and Validation Automatic transformations of *mograms* play a decisive role in compilation and MDE. Such transformations can be the translation from one language to another (*e.g.*, code generation targeting a specific platform) or *mogram* refactorings (*e.g.*, code optimization). As for any other automated task, there is a need to ensure that some structural and behavioral properties are preserved by successive transformations. Hence we consider V&V as a key concern of SLE. It requires the integration of formal methods while remaining as transparent as possible to domain experts who define languages and transformations.

There are of course several benefits in using DSLs rather than general-purpose programming languages with respect to V&V. V&V tools often face intractable or even undecidable

problems due to the high expressivity of general-purpose languages. DSLs can be engineered to be less expressive than general-purpose programming languages, using a reduced number of domain-specific concepts. This reduced expressivity makes it possible to use V&V tools. Typically, the reduced number of concepts of a DSL implies a reduced size of the input domain to be covered by analyses.

DSLs also come with separation of concerns, enabling the division of large and complex systems into several smaller pieces (*i.e.*, into several domain-specific *mograms* rather than one program). Such small *mograms* allow the use of V&V tools (*e.g.*, model-checking) which would not scale on complete systems.

However the use of DSLs instead of general-purpose languages raises new problems and needs. Widely used general-purpose languages can rely on a huge number of users writing different kinds of *mograms*, which provides many test cases for the compilers (*e.g.*, gcc). DSLs by definition do not have such a wide user base, implying the need to formally assess the DSL tooling, and to automate the V&V tasks.

V&V of transformations to ensure preservation of properties has been explored by the CompCert project [75] which aims at providing an entirely verified optimizing C compiler and which is based on several DSLs such as CLight (a subset of C) or a language for Linear Temporal Logic and on a memory model designed to be formally analysed [76]. Several other authors address the verification of model transformations using *bi-simulation* of input and ouput models [77, 78] or graph transformation rules [79]. However, these transformations are still manually verified by their developers.

Automated testing of model transformations is another way to ensure property preservation for transformations which cannot or must not be entirely verified. Baudry *et al.* [80] have identified barriers to such an automated testing, namely the inherent complexity of the graph structures manipulated by the transformations, the lack of maturity of model management environment and the heterogeneity of transformation languages and techniques.

5 Conclusion and Perspectives

According to Hutchinson *et al.* [81], one of the main successful uses of MDE in industry is in the design and implementation of DSLs, each specially built to handle a given concern. We should probably better account for this fact when we teach MDE to both engineers and students. This article can be seen as going one step further. Our collective experience is that both communities – MDE and compilation – have a lot to gain in a better understanding of the other side's experience and technologies. We hope that this article will raise the interest of both communities in bridging the chasm, making, on one hand, MDE more efficient and, on the other hand, allowing compiler technology to better leverage software engineering and to meet in a new *Software Language Engineering*.

While SLE is becoming a key concern in software engineering, some challenges should be tackled by relying on MDE and compilation expertise. Nevertheless, some other challenges cannot be solved directly by synergies between MDE and compilation, and should also benefit from other fields of computer science (*e.g.*, formal methods).

References

1. Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation (PLDI '89)*, pages 131–145. ACM, 1989.
2. Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 625–634. IEEE, 2004.
3. Paul Klint, Jurgen J. Vinju, and Tijs van der Storm. Language design for meta-programming in the software composition domain. In Alexandre Bergel and Johan Fabry, editors, *Proceedings of the 8th International Conference on Software Composition (SC '09)*, number 5634 in Lecture Notes in Computer Science, pages 1–4. Springer, 2009.
4. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.
5. James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
6. OMG. *Unified Modeling Language (UML) 2.1.2 Superstructure*, 2007.
7. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In Lionel C. Briand and Clay Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS/UML '2005)*, number 3713 in Lecture Notes in Computer Science, pages 264–278. Springer, 2005.
8. Simon Pickin, Claude Jard, Thierry Jeron, Jean-Marc Jézéquel, and Yves Le Traon. Test Synthesis from UML Models of Distributed Software. *IEEE Transactions on Software Engineering*, 33(4):252–269, 2007.
9. Pierre-Alain Muller, Frédéric Fondement, Franck Fleurey, Michel Hassenforder, Rémi Schnekenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model Driven Analysis and Synthesis of Textual Concrete Syntax. *Journal of Software and Systems Modeling*, 7(4):423–442, 2008.
10. OMG. *Meta Object Facility (MOF) 2.0 Core Specification*, 2006.
11. Jean-Marc Jézéquel. Model driven design and aspect weaving. *Journal of Software and Systems Modeling*, 7(2):209–218, 2008.
12. Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
13. Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, 2007.
14. Donald E. Knuth. On the translation of languages from left to rigth. *Information and Control*, 8(6):607–639, 1965.

15. Donald E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971.
16. Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
17. T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
18. Pierpaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. *Software: Practice and Experience*, 25(6):657–679, 1995.
19. Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, 1998.
20. Matteo Risoldi and Didier Buchs. A domain specific language and methodology for control systems GUI specification, verification and prototyping. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '07)*, pages 179–182. IEEE, 2007.
21. Christian Hahn. A domain specific modeling language for multiagent systems. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *Proceedings of 7th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS '08)*, pages 233–240. IFAAMAS, 2008.
22. OMG. *UML Object Constraint Language (OCL) 2.0 Specification*, 2003.
23. OMG. *MOF 2.0 Query/ View/ Transformation (QVT) Specification*, 2008.
24. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In Andrew W. Appel and Alex Aiken, editors, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 105–118. ACM, 1999.
25. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
26. D.S. Wile. *POPART: Producer of Parsers and Related Tools, System Builder's Manual.* USC/Information Sciences Institute, 1982.
27. Dominique Clement, Janet Incerpi, and Gilles Kahn. CENTAUR: towards a software tool box for programming environments. In Fred Long, editor, *Proceedings of the International Workshop on Software Engineering Environments (SEE '90)*, number 467 in Lecture Notes in Computer Science, pages 287–304. Springer, 1990.
28. Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Rewriting Strategies in Java. *Electronic Notes in Theoretical Computer Science*, 219:97–111, 2008.
29. Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*, pages 194–206. ACM, 1973.
30. Florian Martin. PAG, - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
31. Jia Zeng, Chuck Mitchell, and Stephen A. Edwards. A Domain-Specific Language for Generating Dataflow Analyzers. *Electronic Notes in Theoretical Computer Science*, 164(2):103–119, 2006.
32. Andrew Stone, Michelle Strout, and Shweta Behere. May/must analysis and the DFAGen data-flow analysis generator. *Information and Software Technology*, 51(10):1440–1453, 2009.
33. Christopher W. Fraser. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, 1991.
34. George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *Proceedings of the 34th annual Design Automation Conference (DAC '97)*, pages 299–302. ACM, 1997.
35. Hristo Nikolov, Todor Stefanov, and Ed F. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, 2008.
36. OMG. *Model Driven Architecture (MDA) Guide, v1.0.1*, 2003.
37. Joao Paulo Almeida, Remco Dijkman, Marten van Sinderen, and Luis Ferreira Pires. On the Notion of Abstract Platform in MDA Development. In *Proceedings of the 8th IEEE International Conference on Enterprise Distributed Object Computing (EDOC '04)*, pages 253–263. IEEE, 2004.
38. Dennis Wagelaar and Viviane Jonckers. Explicit Platform Models for MDA. In Lionel C. Briand and Clay Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS '05)*, number 3713 in Lecture Notes in Computer Science, pages 367–381. Springer, 2005.
39. Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *Proceedings of the Future of Software Engineering Symposium (FOSE '07)*, pages 37–54. IEEE, 2007.
40. D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
41. Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, *Proceedings of Satellite Events at the MODELS 2005 Conference (WiSME '05)*, number 3844 in Lecture Notes in Computer Science, pages 159–168. Springer, 2005.
42. Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
43. Asma Charfi, Chokri Mraidha, Sébastien Gérard, François Terrier, and Pierre Boulet. Toward optimized code generation through model-based optimization. In *Proceedings of the 13th Design, Automation and Test in Europe Conference (DATE '10)*, pages 1313–1316. IEEE, 2010.
44. Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG - A Generator for Efficient Back Ends. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI '89)*, pages 227–237. ACM, 1989.
45. Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.
46. Antoine Floch, Tomofumi Yuki, Clément Guy, Steven Derrien, Benoit Combemale, Sanjay Rajopadhye, and Robert France. Model-Driven Engineering and Optimizing Compilers: A bridge too far? In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS '11)*, number 6981 in Lecture Notes in Computer Science, pages 608–622. Springer, 2011.
47. Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

48. Eugene Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '85)*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer, 1985.
49. Jean-Marie Favre, Dragan Gasević, Ralf Lammel, and Andreas Winter. Guest Editors' Introduction to the Special Section on Software Language Engineering. *IEEE Transactions on Software Engineering*, 35(6):737–741, 2009.
50. Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. In *Proceedings of the 3rd European Congress on Embedded Real Time Software (ERTS '06)*, 2006.
51. Hubert Garavel and Frédéric Lang. NTIF: A General Symbolic Model for Communicating Sequential Processes with Data. In Doron Peled and Moshe Y. Vardi, editors, *Proceedings of the 22nd International Conference on Formal Techniques for Networked and Distributed Sytems (FORTE '02)*, number 2529 in Lecture Notes in Computer Science, pages 276–291. Springer, 2002.
52. Bernard Berthomieu, Pierre-Olivier Ribet, François Vernadat, J. L. Bernartt, Jean-Marie Farines, Jean-Paul Bodeveix, Mamoun Filali, Gérard Padiou, Pierre Michel, Patrick Farail, Pierre Gauffilet, Pierre Dissaux, and Jean-Luc Lambert. Towards the verification of real-time systems in avionics: the Cotre approach. *Electronic Notes in Theoretical Computer Science*, 80:203–218, 2003.
53. Robby, Matthew B. Dwyer, and John Hatcliff. Domain-specific model checking using the bogor framework. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pages 369–370. IEEE, 2006.
54. Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS '08)*, pages 1–8, 2008.
55. Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schoenboeck, and Wieland Schwinger. Fact or Fiction - Reuse in Model-to-Model Transformations. In *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT '12)*, Lecture Notes in Computer Science. Springer, 2012.
56. Arnaud Cuccuru, Chokri Mraidha, François Terrier, and Sébastien Gérard. Templatable metamodels for semantic variation points. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *Proceedings of the 3rd European conference on Model driven architecture-foundations and applications (ECMDA-FA'07)*, number 4530 in Lecture Notes in Computer Science, pages 68–82. Springer, 2007.
57. Juan de Lara and Esther Guerra. From types to type requirements: genericity for model-driven engineering. *Journal of Software and Systems Modeling*, 2011.
58. Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of the 7th International Conference on the Unified Modeling Language (UML '04)*, number 3273 in Lecture Notes in Computer Science, pages 290–304. Springer, 2004.
59. Mickael Kerboeuf and Jean-Philippe Babau. A DSML for reversible transformations. In Cristina Videira Lopes, editor, *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling*. ACM, 2011.
60. Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic Model Refactorings. In Andy Schürr and Bran Selic, editors, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, number 5795 in Lecture Notes in Computer Science, pages 628–643. Springer, 2009.
61. Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Cuadrado, Esther Guerra, and Juan de Lara. Reusing model transformations across heterogeneous metamodels. In *Proceedings of the International Workshop on Multi-Paradigm Modeling*. Online Publication, 2011.
62. Kim B. Bruce and Joseph C. Vanderwaart. Semantics-Driven Language Design:: Statically Type-Safe Virtual Types in Object-Oriented Languages. *Electronic Notes in Theoretical Computer Science*, 20(0):50–75, 1999.
63. Erik Ernst. Family Polymorphism,. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, number 2072 in Lecture Notes in Computer Science, pages 303–326. Springer, 2001.
64. Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling*, 6(4):401–413, 2007.
65. Clément Guy, Benoit Combemale, Steven Derrien, and Jean-Marc Jézéquel. On Model Subtyping. In Antonio Valecillo, editor, *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA '12)*, number 7349 in Lecture Notes in Computer Science, pages 400–415. Springer, 2012.
66. Andrés Vignaga, Frédéric Jouault, María Bastarrica, and Hugo Brunelière. Typing artifacts in megamodeling. *Journal of Software and Systems Modeling*, 10(1):1–15, 2011.
67. Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. *SIGPLAN Notices*, 10(6):114–121, 1975.
68. Regina Hebig, Andreas Seibel, and Holger Giese. On the Unification of Megamodels. In Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Laszlo Lengyel, Tiziana Magaria, Julia Padberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Workshop on Multi Paradigm Modeling at the MODELS 2010 Conference (MPM '10)*, volume 42 of *Electronic Communications of the EASST*, pages 1–13. EASST, 2010.
69. Jean-Marie Favre and Tam Nguyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):59–74, 2005.
70. Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying overlaps of heterogeneous models for global consistency checking. In Jürgen Dingel and Arnor Solberg, editors, *Proceedings of the First International Workshop on Model-Driven Interoperability (MDI '10)*, number 6627 in Lecture Notes in Computer Science, pages 42–51. Springer, 2010.
71. R. Salay, J. Mylopoulos, and S. Easterbrook. Managing Models through Macromodeling. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*, pages 447–450. IEEE, 2008.
72. Esther Guerra, Juan de Lara, and Fernando Orejas. Pattern-Based Model-to-Model Transformation: Handling Attribute

Conditions. In Richard F. Paige, editor, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT '09)*, number 5563 in Lecture Notes in Computer Science, pages 83–99. Springer, 2009.

73. Mickael Clavreul, Olivier Barais, and Jean-Marc Jézéquel. Integrating legacy systems with MDE. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*, pages 69–78. ACM, 2010.
74. Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. Inter-modelling: from theory to practice. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of the 13th International Conference on Model driven engineering languages and systems: Part I (MODELS '10)*, number 6394 in Lecture Notes in Computer Science, pages 376–391. Springer, 2010.
75. Xavier Leroy. Formal verification of an optimizing compiler. In Franz Baader, editor, *Proceedings of the 18th International Conference on Term Rewriting and Applications (RTA'07)*, number 4533 in Lecture Notes in Computer Science, pages 1–1. Springer, 2007.
76. Xavier Leroy and Sandrine Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.
77. Anantha Narayanan and Gabor Karsai. Towards Verifying Model Transformations. *Electronic Notes in Theoretical Computer Science*, 211:191–200, 2008.
78. Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, 2009.
79. Hartmut Ehrig and Claudia Ermel. Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations (ICGT '08)*, number 5214 in Lecture Notes in Computer Science, pages 194–210. Springer, 2008.
80. Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.
81. John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 471–480. ACM, 2011.