Model-Driven Engineering and Optimizing Compilers: A bridge too far?

Antoine Floch, Tomofumi Yuki, Clément Guy, Steven Derrien, Benoit Combemale, Sanjay Rajopadhye, Robert France

▶ To cite this version:

HAL Id: inria-00613575

https://hal.inria.fr/inria-00613575

Submitted on 4 Aug 2011

Model-Driven Engineering and Optimizing Compilers: A bridge too far?

Antoine Floch[1], Tomofumi Yuki[2], Clement Guy[1], Steven Derrien[1], Benoit Combemale[1], Sanjay Rajopadhye[2], and Robert B. France[2]

[1] University of Rennes 1, IRISA, INRIA
{antoine.floch,clement.guy,steven.derrien,benoit.combemale}@irisa.fr
[2] Colorado State University
{yuki,svr,france}@cs.colostate.edu

Abstract. A primary goal of Model Driven Engineering (MDE) is to reduce the cost and effort of developing complex software systems using techniques for transforming abstract views of software to concrete implementations. The rich set of tools that have been developed, especially the growing maturity of model transformation technologies, opens the possibility of applying MDE technologies to transformation-based problems in other domains.

In this paper, we present our experience with using MDE technologies to build and evolve compiler infrastructures in the optimizing compiler domain. We illustrate, through our two ongoing research compiler projects for C and a functional language, the challenging aspects of optimizing compiler research and show how mature MDE technologies can be used to address them. We also identify some of the pitfalls that arise from unrealistic expectations of what can be accomplished using MDE and discuss how they can lead to unsuccessful and frustrating application of MDE technologies.

1 Introduction

Model Driven Engineering (MDE) research is primarily concerned with reducing the accidental complexities associated with developing complex software systems [1]. This is accomplished through the use of technologies that support rigorous analysis and transformation of abstract descriptions of software to concrete implementations [2]. At the core of MDE are modeling languages that are typically defined as metamodels. The metamodels are expressed in a metalanguage such as the OMG Meta-Object Facility (MOF). Developers can use these modeling languages to describe complex systems at multiple levels of abstraction and from a variety of perspectives. MDE is essentially concerned with transforming descriptions of software artifacts to other forms that better serve specific purposes. For example, MDE techniques can be used to transform a detailed design model expressed in the Unified Modeling Language (UML) [3] to a Java program that can be compiled and executed, or to transform an abstract description of software to a performance model that can be used to estimate software performance characteristics.

It may seem that researchers in the MDE and optimizing compilers communities tackle vastly different problems. However, some of the more mature MDE technologies

can be fruitfully leveraged in research-oriented optimizing compiler infrastructures. The connection between MDE and the optimizing compiler domains stems from the observation that the intermediate representations of optimizing compilers are *abstractions* of input programs that are repeatedly *transformed* to more efficient forms. In addition, researchers in the optimizing compiler domain need tools that enable rapid development and continuous evolution of compiler implementations built specifically to prototype and evaluate research ideas. The preceding concerns makes the application of MDE techniques in the optimizing compiler research domain appealing and useful.

In this paper, we illustrate the role MDE techniques can play in the optimizing compiler research domain using two on-going research compiler projects as case studies. These two compilers accept and optimize significantly different languages, C/C++ and a purely functional language, but they both benefit from the use of MDE techniques in a similar manner. We identify significant tasks in research compiler development, and highlight how MDE techniques can help reduce the cost and effort of performing these tasks. Our experience provides some evidence that bridging the two communities is possible and can yield significant benefits. Unrealistic expectations of what MDE can do may lead to ineffective use of MDE in the optimizing compiler domain, and thus we discuss pitfalls that users should be aware of when using these techniques. Our experience also revealed that the concept of a *transformation* in the optimizing compiler domain is broader than the concept currently supported by MDE tools. It would be interesting to explore how the broader notions of transformation can be leveraged in the MDE community. In this paper we identify some of the broader transformation concepts that may usefully be explored by the MDE community.

The rest of this paper is organized as follows. In Section 2, we briefly characterize optimizing compiler research, and the similarity between compiler intermediate representations and models. Then we describe common challenges that arise in research compiler development in Section 3. Section 4 highlights the benefits of applying MDE techniques to compilers through examples taken from our research compilers being developed with MDE. In Section 5, we present some of the pitfalls that researchers in the optimizing compiler domain need to be aware of in order to use MDE effectively. Finally, we give our conclusions and perspectives in Section 6.

2   Optimizing Compilers

Experimental compiler infrastructures play an important role in compiler research. Such infrastructures are different from production compilers as illustrated by our two example infrastructures, namely GeCoS[3] and AlphaZ[4].

2.1   Optimizing Compiler Research

Optimizing compiler research infrastructures are key elements in many research communities including High Performance Computing [4] and Embedded Systems Design

---

[3] http://gecos.gforge.inria.fr
[4] http://www.cs.colostate.edu/AlphaZ/

Automation [5]. Optimizing compilers aim at obtaining the best possible performance from input programs. Performance is to be understood in its broader sense, and may either correspond to execution time, power/energy consumption, code size or any combination of these metrics. Achieving better performance depends on both the target application and the target architecture.

Indeed, target physical machines can be very different, ranging from general purpose single/multi-core processors to special purpose hardware. Such target machines include, Graphics Processing Units (GPUs), Application Specific Instruction-set Processors (ASIPs) and/or application specific hardware accelerators implemented on either Complex Programmable Logic Devices (CPLD) or dedicated VLSI circuits.

For example, in High Performance Computing, applications (climate modeling, weather prediction, physical simulation, etc.), often run on supercomputers, where the execution time directly affects the cost. In Embedded Systems Design Automation, many applications involve Digital Signal Processing and/or multimedia algorithms, and target machines usually consist of special purpose hardware with a short life cycle. For these targets, the focus is cost (i.e., silicon area), performance/energy trade-off and also design time (because of time to market constraints).

Optimizing compiler research is therefore a collection of efforts to achieve high performance by analysis and optimizations of programs at the compiler level. Individual research usually tackles very specific problems (automatic parallelization, instruction selection, etc.), and can be therefore be seen as developing building blocks of a full compiler. As a consequence, significant effort is spent on prototyping new analysis/transformation passes in a research compiler infrastructure. There is hence a strong need for highly productive compiler infrastructure, where research ideas and prototypes can be quickly validated.

2.2 Optimizing Compiler Infrastructures

Compilers range from industrial strength production compilers to experimental ones that tackle domain specific problems and generally require more user intervention and/or multiple input specifications. However, most compilers share the same structure with three stages:

1. **Parsing** takes some form of input, usually a program written in a textual language, and constructs its Intermediate Representation (IR). The input language can be virtually anything, from complex languages such as C++ to domain specific languages. Similarly, compiler IRs range from Abstract Syntax Trees to complex data structures including additionnal information (typing, control flow, etc.).
2. Program **Optimizations** are repeatedly performed as transformations on the IR. The result of a transformation may stay in the same IR, or it may be another (generally lower level) IR, better suited to support platform specific optimizations. In the context of optimizing compilers, these stages involve complex combinatorial optimizations problems.
3. **Code Generation** translates the transformed IR to either executable binaries or source programs (which may not use the same language). As a matter of fact, Source-to-Source compilers are very common in optimizing compiler research.
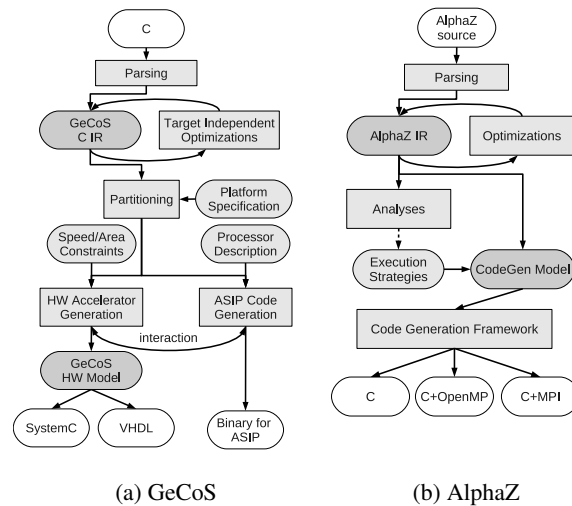
(a) GeCoS   (b) AlphaZ

Fig. 1: Basic flow of GeCoS and AlphaZ. Transformations are performed in various places for optimization or lowering the level of abstraction. There are a number of different specifications that are given as additional inputs, and multiple different outputs are produced.

2.3 Role of Compiler Infrastructures

Research on optimizing compiler tries to answer questions such as: what transformations leads to more efficient code, how to define good cost functions to predict performance, how to ensure legality of a transformation, etc.

Whenever a new and/or better answer to one of these questions is found, researchers provide a "proof of concept" implementation of their approach that is used to experimentally validate their claims. The main role of research oriented compiler infrastructures is therefore to facilitate such rapid prototyping. As a consequence, the infrastructure code base tends to evolve very quickly. Fortunately, such compilers are not expected to be as stable and robust as production compilers. Similarly, the performance of the compiler implementation itself is rarely an issue, as long as the compiler output ultimately leads to improved performance.

2.4 GeCoS and AlphaZ

We now present two research compiler infrastructures: GeCoS and AlphaZ that illustrate the diversity of the optimizing compiler infrastructure landscape.

GeCoS is a C compiler infrastructure geared toward embedded system design. It can be used for Application Specific Processors (ASIPs) design and Custom Hardware Accelerator Synthesis. It can be used as a Source-to-Source C compiler or as a standalone flow with a complete retargetable compiler back-end and support for hardware synthesis via back-end to generate hardware descriptions.

AlphaZ is a system for exploring and prototyping analyses, transformations, and code generations for a class of programs that fit a formalism called the Polyhedral model. The system takes programs written in an equational language as inputs, and produces C codes targeting general purpose processors, in particular, multi-core architectures.

It is important to note that these two compilers take very different input languages, C and an equational language, and also produce outputs for diverse set of target platforms, from custom application specific hardware accelerators to general purpose processors. The internal flow of the two compilers are also significantly different as depicted in Figure 1. GeCoS has a number of different IRs being used at different stages of compilation, but AlphaZ performs all optimizing transformations in a single IR.

Interestingly, and despite all these differences, the developers of these compilers face very similar issues and challenges, that we describe in the following section.

3 Challenges in Optimizing Compilers

Research compiler infrastructure developers face many challenges, some of them being quite specific to compiler design, as explained below.

3.1 Maintainable and Sustainable Code

One of the fundamental challenges in our context is sustainability and maintainability of the code. Research-oriented compilers are very complex pieces of software, generally developed by generations of graduate students and interns working on parts of the infrastructure. This high turn over rate raises the need to support incremental development practices, and the seamless homogenization of programming style.

Furthermore, it is difficult to expect that contributors to such compiler infrastructures have a solid software engineering background and/or practice. Worse, many students working in the embedded system design automation community have an electrical/computer engineering background, rather than computer science.

3.2 Structural Validity of Intermediate Representation

When writing an optimizing transformation, one of the most tedious task consists in making sure that the transformed IR remains consistent with respect to the IR data structure. There are many consistency rules that must be enforced by the IR. For example, in many imperative programs, the use of a variable in a statement must be preceded by its declaration somewhere in the program execution flow. Such validations are generally performed on the IR after parsing. In research compiler infrastructures, it may also be desirable to perform these checks after a call to a transformation/optimization so as to spot obvious inconsistencies as early as possible. Writing these static checks is however very time consuming, as it involves a lot of navigation and book keeping operations, which are tedious to write and very error prone.

Moreover, experimental languages are more frequently extended and/or modified than conventional languages. Any non-trivial extensions or modification of the language

forces developers to spend significant effort for updating these analysis, making this task even more time consuming.

3.3   Complex Querying of the IR

A compiler optimization is generally only applicable to a narrow subset of constructs of the language, and for which a precise set of preconditions holds. Retrieving the target constructs and checking that the corresponding preconditions are enforced requires a lot of querying within the IR. Many of these queries actually correspond to more or less simple pattern matching operations. As an example, a simple loop unrolling transformation requires to retrieve all the loop constructs from the IR in which the bounds and the step are constant. Then, for each of such loop, the transformation must check that the loop body has no side-effect on the loop iterator.

While navigation can be efficiently handled through the use of visitor design patterns, the code complexity induced by the query implementation quickly makes the code difficult to understand and to maintain.

3.4   Interfacing with External Tools

Experimental research compilers infrastructures make an extensive use of third party libraries that are used for very specific purposes. For example, boolean satisfiability, integer linear programming solvers and/or machine learning libraries are often used to express and solve compiler optimization problems. These libraries may be implemented in various languages, and therefore require custom bindings if the compiler is written in a different language.

Similarly, there also exist powerful tools to ease the implementation of complex pattern matching operations over trees/graphs. For example, Tom/Gom[5] provides a term rewriting engine, particularly well suited to express compiler optimization. However, exposing the IR to the Tom/Gom engine requires a complex mapping specification that has to be written by hand.

3.5   Semantics Preserving Transformations

One of the most fundamental requirements for a compiler is to ensure that the semantics of the original source code are retained by the output. This has led to growing emphasis on provability, as seen in the CompCert [6] project that implemented a verified production compiler, as well as increasing influence of theorem provers in compilers [7]. In the context of an optimizing compiler, where usually only small parts of the code are changed based on specific preconditions, an important challenge is that in addition to proving that a proposed transformation preserves the original semantics, we must also prove that the *implementation* of the transformation correctly preserves the designer's intention. Ideally, every transformation needs a proof of correctness before being implemented, and tools to certify that the implementation preserves this. Such an ability would turn out to be very useful to identify mismatch between theory and practice (unsupported corner cases, flawed algorithm, etc.)

---

[5] http://tom.loria.fr/

3.6 Systematic Approaches for Capturing Domain Specific Knowledge

Optimizing compilers often fail at fully taking advantage of all optimization opportunities because they lack of knowledge about low level details of the target machine.

These limitations have been addressed by proposing language dialects and/or extensions to address the shortcoming of existing general purpose programming languages. This is particularly true for parallel programming where such domain specific knowledge is mandatory to achieve reasonable performance: [8,9,10]. Other approaches advocate the use of alternatives and/or more specific languages, that better fit some purpose. AlphaZ uses an equational language as inputs, where the computation is specified as mathematical equations. One of the motivations for this equational language is the separation of concerns; what to compute should be separated from other choices, such as memory allocation.

Such domain specific knowledge may also be used by the compiler developers themselves. For example, most compiler frameworks rely on a formal description of the processor instruction set and of its micro-architecture. This description is then used to automate the porting of the compiler to that new architecture.

However, the use of DSL in the context of optimizing compiler is hindered by by the fact designing custom languages involves high development and maintenance efforts. In particular, even if existing tools (e.g., ANTLR, Yacc) help addressing parsing issues, they fail at providing facilities for interfacing the parser output to the target IR (and to other components of the compiler). This is a significant problem in compiler research, where domain specific languages are generally designed incrementally, and where adding a new feature in the language has hence significant development cost.

Of course, these problems are even more severe when it comes to extending GPL with embedded DSLs, as the languages that have to be extended are often very complex (e.g., C/C++). As of now, even compiler compilers do not provide enough facilities to help solving this type of problems.

3.7 Code Generation

In a research context, compilers may need to target multiple architectures or languages. In addition, a same input program can lead to several distinct code in the same target language (e.g., sequential C code and MPI parallel C code). This is particularly common in research paralleling compilers that deals with emerging architectures (e.g., IBM/SONY/Toshiba Cell BE, GPGPUs, Intel Larrabee) to explore optimization opportunities. Developing code generators for each target and/or language requires a lot of effort that could be significantly reduced by the use of facilities to reuse and customize code generators.

4 How to Use MDE in Compilers

Since compiler IRs are abstractions used to represent programs, they are by essence models (an instance of IR is an abstraction of the given source code). In this context, the grammar of the source language, or more often the structure of IR, becomes the

metamodel. We now report how three kind of MDE uses can answer to the challenges described in Section 3. The description leverages our development experience after we started using these technologies two years ago for both the GeCoS and AlphaZ infrastructures.

4.1 Direct MDE Benefits

The GeCoS and AlphaZ compilers started from a significant legacy code base, and we therefore soon felt a strong need for a formalized and standardized software development process. Because the GeCoS compiler infrastructure was already tightly coupled with the Eclipse environment, it seemed natural for us to use the metamodeling facilities provided by EMF[6].

Model can Serve as a Documentation An immediate benefit is that all the key information lies in the metamodel specification. It focuses on the problem domain, without excessive implementation details and helps bootstrap new developers into a project, even when documentation is lacking.

Code Generator Homogenization and good development practices are some of the most immediate benefits of MDE. The use of code generators (e.g., generic EMF Java code generator) providing standardized interfaces and ensuring (structural) model consistency has a direct impact on code quality. The advanced reflexivity (e.g., containments and structural features) of the generated code eases the development of tool functions without requiring tedious instrumentation (that is usually far from the process being modelled) of the metamodel.

Generic Tools All the generic tools based on the model specification also offer significant added value at zero development cost. These tools can help a lot to increase the robustness of compilers. First, a model enforces its metamodel simple structural properties (arity of references and containments consistency). These properties can be easily verified by using standard serialization process.

Moreover, the EMF Tree editor generated from the metamodel specification also proved to be helpful. During early development stages it helped us in fixing bugs through an understandable visualization of transformation results. Figure 2 shows the slightly customized editors for AlphaZ and GeCoS IRs.

Finally, Object Constraint Language (OCL) can be used to express additional invariant rules (and pre/post-conditions) that are checked at runtime against model instances. This turns out to be particularly useful in the context of an optimizing compiler, as it helps ensure that a given transformation preserves the correctness of transformed IR. For example, many transformations requires the input IR to be in SSA[7] form. A simple OCL query can easily check this property as a post-condition of the SSA transformation and as pre-condition of the optimizations.
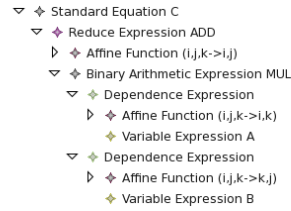
---

[6] http://www.eclipse.org/modeling/emf/

[7] Static Single Assignment. All variables within a function are assigned exactly once

```
affine MMM {P, Q, R|P>0 && Q>0 && R>0}
given
    float A {i,k| 0<=i<P && 0<=k<Q};
    float B {k,j| 0<=k<Q && 0<=j<R};
returns
    float C {i,j| 0<=i<P && 0<=j<R};
through
    C[i,j] = reduce(+,[k], A[i,k]*B[k,j]);
.
```
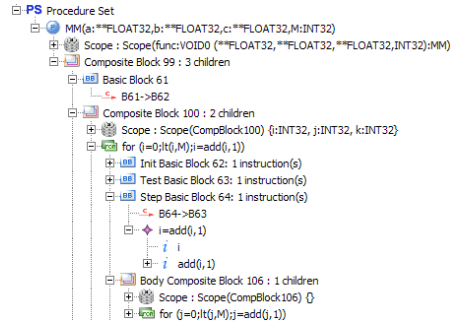
(a) AlphaZ source

```
▽ ◆ Standard Equation C
  ▽ ◆ Reduce Expression ADD
    ▷ ◆ Affine Function (i,j,k->i,j)
    ▽ ◆ Binary Arithmetic Expression MUL
      ▽ ◆ Dependence Expression
        ▷ ◆ Affine Function (i,j,k->i,k)
          ◆ Variable Expression A
      ▽ ◆ Dependence Expression
        ▷ ◆ Affine Function (i,j,k->k,j)
          ◆ Variable Expression B
```

(b) AlphaZ IR

```
void MM(
        float **a,
        float **b,
        float **c,
        int M) {
    int i,j,k;
    for (i=0;i<M;i=i+1) {
        for (j=0;j<M;j=j+1) {
            c[i][j]=0.0;
            for (k=0;k<M;k=k+1) {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
```

(c) C source

```
⊟·PS Procedure Set
  ⊟·◯ MM(a:**FLOAT32,b:**FLOAT32,c:**FLOAT32,M:INT32)
    ⊞·▦ Scope : Scope(func:VOID0 (**FLOAT32,**FLOAT32,**FLOAT32,INT32):MM)
    ⊟·▦ Composite Block 99 : 3 children
      ⊟·▦ Basic Block 61
        └─ B61->B62
      ⊟·▦ Composite Block 100 : 2 children
        ⊞·▦ Scope : Scope(CompBlock100) {i:INT32, j:INT32, k:INT32}
        ⊟·▦ for (i=0;lt(i,M);i=add(i,1))
          ⊞·▦ Init Basic Block 62: 1 instruction(s)
          ⊞·▦ Test Basic Block 63: 1 instruction(s)
          ⊟·▦ Step Basic Block 64: 1 instruction(s)
            └─ B64->B63
            ⊟·◆ i=add(i,1)
              ├─ i  i
              ⊞·i  add(i,1)
          ⊟·▦ Body Composite Block 106 : 1 children
            ⊞·▦ Scope : Scope(CompBlock106) {}
            ⊞·▦ for (j=0;lt(j,M);j=add(j,1))
```

(d) GeCoS IR

Fig. 2: Matrix multiplication in AlphaZ and C, and its corresponding intermediate representation in AlphaZ and GeCoS.

4.2 Using Metatools

MDE can significantly increase efficiency through generic tools targeted at complex and specific development tasks.

Facilities to Define DSLs As discussed in Section 3, compilers can benefit a lot by capturing domain specific knowledge. Tools such as Xtext[8] or EMFText[9] can provide concrete textual syntax to a DSL, together with an editor with basic syntax highlighting and auto-completion.

For example, the equational language used in AlphaZ is parsed using Xtext generated parser. Because the language is experimental, minor/major language changes or extensions occur frequently, and the use of model based tools makes it easier to maintain the consistency between the parser and other components.

---

[8] http://www.eclipse.org/Xtext/
[9] http://www.emftext.org/

Facilities to Generate Code Model-to-Text (M2T) tools such as Xpand/Xtend[10] provides a modular and extensible template based specification of the generated text through imports and aspects. In Xpand, each template rule supports parametric polymorphism that simplifies the management of specialized entities (especially useful for compiler IRs described as an abstract syntax tree with specialized nodes).

In both GeCoS and AlphaZ, these facilities are heavily utilized for code generation. In the case of GeCoS, the compiler IR is eventually translated into a model that represents the target hardware to generate, and different templates are used to generate VHDL (hardware description language), or SystemC (C-like language for high-level synthesis). In AlphaZ, the compiler IR corresponding to the functional representation is transformed into another representation that is closer to imperative programs. From the imperative IR, Xpand aspects are used to generate variations such as C code with OpenMP pragmas for loop parallelization, or alternate implementations of multidimensional arrays.

4.3   Defining Metatools

All metamodels are described using the same model (metametamodel), and thus we can manipulate/analyze metamodels in a generic fashion, by developing in the metametamodel. One of the benefits, is to bind some generic behaviors to the manipulated metamodels. This can be achieved by a generative approach or even by interpretation for a fast prototyping. In both cases, a dedicated environment based on the common metametamodel provides a language to describe the executable behaviors.

Generative Approaches The definition of generative metatools allows the developers to automate a task to all or some subset of metamodels comforting to the metametamodel. Some tasks may be fully automated through Model-to-Model (M2M) transformations on the metamodel using tools such as ATL[11] or Kermeta[12]. Others, can be guided by DSLs. The resulting metatools generate codes corresponding to the tasks instantiated for different metamodels.

Structural software design patterns are a perfect example of generic concepts. Expressing them at a metametamodel level gives a powerful toolbox to the developers who can apply or reuse these patterns on all their metamodels. In compilers, we need to query/transform the IR. This is done mostly by using the visitor design pattern and extensively used tree traversal algorithms such as depth first and breadth first. Whereas adding a visitor pattern to an existing code is tedious since it needs to add a function to each visited entity, it can be done automatically using a simple M2M transformation. Behavior codes of the various traversal strategies are inferred by a simple containment analysis and added as annotations to the transformed metamodel.

Using DSLs proves to be especially useful to build generative tools for more complex repeated tasks such as interfacing with external tools. We give here two examples

---

[10] http://www.eclipse.org/modeling/m2t/?project=xpand

[11] http://www.eclipse.org/atl/

[12] http://www.kermeta.org/

of DSL-based metatools that significantly enhanced our productivity for difficult and time consuming tasks.

Example : Graph Mapper Compiler optimizations often rely on graph-based IR and thus can benefit from an external, optimized graph library implementation. The key idea of *Graph Mapper* is to map the library graph implementation to the IR instead of defining the IR from this graph (through inheritance). Hence, we designed a DSL that takes any metamodel as input and helps to explicitly describes how to map nodes and edges to this metamodel. The tool then generates an adapter to the external graph implementation.

Example : Tom/Gom bindings Tom/Gom is a term rewriting system for Java. Both GeCoS and AlphaZ use Tom/Gom for a number of transformations that are pure rewriting of the IR. For example, expressions that occur in programs like $2i + 2i$ can be simplified as $4i$ by applying the rewrite rule expressed as the following:

$$\texttt{add}(\texttt{term}(\texttt{c1}, \texttt{var}), \texttt{term}(\texttt{c2}, \texttt{var})) \rightarrow \texttt{term}(\texttt{c1} + \texttt{c2}, \texttt{var})$$

that simplifies additions of two terms when the variables in the two linear terms are identical.

These rules are much easier to express and to understand than visitor based implementations. However, Tom/Gom requires bindings from expressions in its language to Java objects. We have developed a tool that automates this task using a simple DSL to specify the terms that are manipulated in a model, and the names of the Tom expressions. Binding specifications are then generated using M2T facilities.

Model Mapping Previous examples of bindings using DSLs can be seen as specific model mappings. Some existing generic mapping languages enable defining such links between metamodels and to use them for M2M transformations. For example the semi-automatic process presented by Clavreul et al. [11] enables defining mappings between two metamodels and generating bidirectional transformations. This kind of approach could be used for external tool interfacing and to pass from one IR to another.

Metamodel Instrumentation Executable metamodeling languages (i.e., action languages provided by the metametamodel) such as Kermeta provide facilities to express executable behaviors directly on the metamodel. They enable the instrumentation of metamodels through aspect oriented modeling thereby providing a very elegant way to achieve separation of concerns. Thus, complex IR transformations can be efficiently described without the need of visitors. Since the metamodel can be instrumented with new attributes and methods in the intent of the transformation, it significantly reduces the complexity of the algorithm implementation.

A concrete example is an M2M transformation from a IR where an instruction is described as a tree to another one where a sequence of instructions corresponds to a directed acyclic graph (DAG). If M2M tools such as ATL provide an easy way of doing M2M for *one to one* mapping rules, it becomes difficult and nearly intractable for a
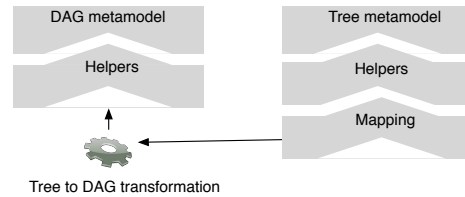
Fig. 3: Tree instructions to DAG instructions transformation through Kermeta. Layers of aspects instrument the metamodels and simplify the transformation code.

standard software engineer to express complex *any to any* mappings. In the case of the tree to DAG transformation it is much more convenient to instrument each of the metamodels by tools aspects and to code the transformation using these new helpful attributes and methods. Some subsets of the added features may be useful for other transformations and can be advantageously split into independent layers as depicted in Figure 3. The helpers layers corresponds to utilities aspects that can be reused in multiple other transformations. The mapping layer adds simple references to avoid the need of expensive maps linking DAG/tree elements.

The instrumentation of a metamodel introduces powerful concepts but also an important tooling overhead in terms of memory and speed. Opening/transforming a large model in a framework such as Kermeta can be quite slow or even impracticable in industrial cases containing tens of thousands of model elements. Although we strongly believe that the use of metamodel instrumentation significantly enhances flexibility and maintainability, the scalability issues we currently are facing prevents us from using this facility to our research compilers.

4.4 Summarizing Answers from MDE to Compilers Challenges

MDE provides low-entry-cost advanced solutions for M2M and M2T transformations making it a very attractive technology for developing compiler components. We now briefly summarize how MDE facilities introduced in this Section correspond to the challenges described in Section 3.

The use of generative programming tools based on metamodel specifications leads to well structured and homogeneous code, and forces programmers to follow good software engineering practices addressing the challenge described in Section 3.1. MDE also contribute to the validation of IR and transformations through enforced simple structural properties and more complex OCL queries, corresponding to challenge in Section 3.2, and partially to challenges in Sections 3.3 and 3.5.

Defining metatools, by generation or instrumentation of the metamodels, simplifies the design of complex transformations/queries on the IR also addressing the challenges in Section 3.3. It also enables the automation of time consuming interfacing with external tools described in Section 3.4. Some of these defined metatools benefits from the facilities for describing DSLs and code generation that also give partial answers to challenges in Section 3.6 and 3.7.

5 Applicability of MDE

While it is possible and beneficial to build bridges that put MDE technologies to work in optimizing compiler research infrastructures, it is also important to understand that unrealistic expectations of MDE approaches can lead to frustration and failure. Some of this may also be due to poor software modeling skills needed to effectively apply MDE.

5.1 Scope

MDE is a very attractive solution to many problems in the complex software system development. For such systems, models can be used to describe the software to be built at various levels of abstraction and MDE technologies can be used to manage the creation and manipulation of the models. Here, models are used to describe solutions to problems. It may be tempting to extrapolate this use of models to system software infrastructures, including optimizing compilers. Indeed, compilers can be viewed as a kind of complex information systems. However, a compiler is much more than an information system. Many of the compiler design challenges involve complex combinational optimization problems that are outside the scope of problems targeted by MDE techniques. Furthermore, many parts (e.g., classic data flow analysis and abstract and non-standard interpretation) require deep understanding of the mathematical foundations of lattice theory, fixed points, etc. As an example, modeling the instruction set of a given processor is not enough to efficiently compile a program for that instruction set. In a sense, modeling the problem is not solving the problem.

A key to effective use of MDE techniques is an understanding that models are created to serve specific purposes, and a good model is one that effectively serves its purposes. Developers need to ensure that the models they build are fit for purpose. For example, a good compiler intermediate model is one that describes a program in a format that can be efficiently analyzed as required in particular compilation stages.

Developers also need to be aware that MDE technologies are not intended to create models that are guaranteed to be fit for purpose. Human creativity is needed to create models that are fit for purpose. MDE techniques are designed to *enhance*, not *replace*, the creative abilities of modelers. They allow modelers to describe and analyze their models in order to build confidence that their models effectively serve its purposes. In the optimizing compiler research, this means that MDE technologies will not help produce better models of programs that are fit-for-use in the compilation process.

5.2 Prerequisites

The first requirement is that team members must be able to deal with abstraction, and more precisely must have solid modeling skills. Modeling here is in its broader sense, from mathematical analytical modeling through combinatorial/operational research optimization problem modeling to UML like approaches.

It is also mandatory for developers to be comfortable with OO programming principles. Our experience has shown that even though most young electrical engineers have followed some OO programming courses in their curriculum, few of them have a good

understanding of its concepts like polymorphism. A basic understanding of design patterns is also required.

To obtain an executable code, the MDE developer needs first to model the software and then to generate it. These two steps introduce a tooling overhead which may slow down the initial development. Moreover, even if most MDE tools provide a low entry cost for common metamodel-based tools (e.g., primary model editor automatically generated from a default configuration of tools, such as EMF, GMF and TMF), the price of a flexible tool is often high software design complexity. If MDE tools rapidly provide prototypes, reaching an industrial level leads to an extra load of understanding.

Developers with experience in complex software development will quickly find MDE attractive. These developers know from their previous experience that the quality of the design is critical, and time spent on modeling may be greater than that for implementation. This family of users will be easily convinced of the interest of metaness. It is not so obvious for inexperienced developers working on simple softwares projects with a low level of flexibility and reusability. Based on our experience, successful use of MDE in the compiler domain requires an open-minded development team that is willing to try software engineering techniques to tackle their development problems. This is often facilitated by an influential champion who is willing to spend the time and effort learning and experimenting with MDE technologies.

6 Conclusion and Perspectives

In this paper, we described how optimizing compiler research and MDE can be easily bridged due to inherent modeling aspects of compilers. We illustrated the benefits of MDE through our experiences in building research compiler infrastructures.

The most obvious benefit is a seamless systematization/homogenization of development practices, something that is often very difficult to achieve in an academic environment. Metamodels also offer an abstract representation of the software, and documents many important design choices. This is a very valuable benefit in a context where most of the development consist in undocumented prototypes. Additionally, metatools and metatooling 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.

Even though MDE has proved to be well suited for solving many of challenges arising in optimizing compilers development, this new context of utilization also raises many open research directions that we believe to be of high interest to the MDE community. First, the growing use of M2M transformations (e.g., to implement compilation passes) raises the need for that ensuring structural and behavioral properties are preserved during model transformations. As a consequence, we see model transformation verification and testing as a very important research challenge that needs to be tackled by the MDE community.

Moreover, since MDE now offers tools that significantly ease the definition of DSLs, it is becoming urgent to efficiently handle their rapid increase in numbers. In particular, a DSL should not be created from scratch if another DSL exists that can be used to

derive the new DSL (e.g., using reutilization and extension), and the DSL tooling (e.g., simulator, checker and generator) should be reused over a family of DSLs. We believe that the ability to capitalize transformations by enabling their application over a family of metamodels rather than on a single metamodel is a very important issue. To address this challenge, we are currently studying a theory leveraging model typing [12] and model mapping [11] so as to be able to manipulate DSLs as first class entities.

Finally, it turns out that applying MDE technologies to the development of optimizing compilers led us to face a scalability barrier. The models manipulated by compilers are indeed generally fairly large (in terms of number of model elements) and are not handled very well by many of the MDE tools. Besides, even if research-oriented optimizing compilers do not suffer from strong constraints on execution time, current MDE technologies renders them unsuitable for industrial strength compilers.

We hope these three topics; semantics preserving transformations, model transformations reuse, and tools scalability; will motivate future work.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In Briand, L., Wolf, A., eds.: Future of Software Engineering 2007. IEEE-CS Press (2007)
2. Schmidt, D.: Guest editor's introduction: Model-driven engineering. Computer **39**(2) (feb. 2006) 25 – 31
3. The Object Management Group: UML 2.0: Superstructure Specification. Version 2.0, OMG, formal/05-07-04 (2005)
4. Hall, M., Padua, D., Pingali, K.: Compiler research: the next 50 years. Communications of the ACM **52**(2) (February 2009) 60–67
5. Tripp, J.L., Gokhale, M., Peterson, K.D.: Trident: From High-Level Language to Hardware Circuitry. IEEE Computer **40**(3) (2007) 28–37
6. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (2006) 42–54
7. Arnold, G., Hölzl, J., Köksal, A., Bodík, R., Sagiv, M.: Specifying and verifying sparse matrix codes. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming, ACM (2010) 249–260
8. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., et al.: Titanium: A high-performance Java dialect. Concurrency Practice and Experience **10**(11-13) (1998) 825–836
9. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: ACM SIGPLAN Notices. Volume 40., ACM (2005) 519–538
10. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications **21**(3) (2007) 291
11. Clavreul, M., Barais, O., Jézéquel, J.M.: Integrating legacy systems with mde. In: ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering and ICSE Workshops. Volume 2., Cape Town, South Africa (May 2010) 69–78
12. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) **6**(4) (December 2007) 401–414