This is the accepted version of a paper presented at the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019.

Citation for the original published paper:

A vision of miking: Interactive programmatic modeling, sound language composition, and self-learning compilation
In: SLE 2019 - Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2019 (pp. 55-60). Association for Computing Machinery, Inc
https://doi.org/10.1145/3357766.3359531

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-272342
A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation

David Broman
KTH Royal Institute of Technology
Sweden
dbro@kth.se

Abstract
This paper introduces a vision of Miking, a language framework for constructing efficient and sound language environments and compilers for domain-specific modeling languages. In particular, this language framework has three key objectives: (i) to automatically generate interactive programmatic modeling environments, (ii) to guarantee sound compositions of language fragments that enable both rapid and safe domain-specific language development, (iii) to include first-class support for self-learning compilation, targeting heterogeneous execution platforms. The initiative is motivated in the domain of mathematical modeling languages. Specifically, two different example domains are discussed: (i) modeling, simulation, and verification of cyber-physical systems, and (ii) domain-specific differentiable probabilistic programming. The paper describes the main objectives of the vision, as well as concrete research challenges and research directions.

CCS Concepts → Theory of computation → Program semantics.

Keywords modeling languages, domain-specific languages, machine learning, compilers, semantics, composition

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE ’19, October 20–22, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6981-7/19/10...
https://doi.org/10.1145/3357766.3359531

1 Introduction
Domain-specific languages (DSLs) can give end users several advantages compared to general-purpose programming languages [58]. In particular, domain-specific problems can be described declaratively at a high level of abstraction, stating what should be solved, rather than explicitly how. Ideally, the DSL compiler environment processes the DSL program automatically and generates an efficient solution with minimal user interaction. In the last decades, domain-specific languages have been used successfully in various domains, such as physical modeling and simulation [15, 41], computer graphics [26], hardware description [1, 6], and probabilistic programming [5, 64].

Although there are several potential benefits with DSLs, the cost of designing a language, developing efficient compilers, and creating user friendly development environments can be very high. Moreover, people with domain knowledge (for instance in biology, mechatronics, or statistics) are typically not programming language or compiler experts. Likewise, compiler experts are seldom domain experts, especially not in several different domains. As a consequence, serious DSL development efforts are substantial undertakings, which can lead to suboptimal solutions, with brittle language semantics or inefficient execution environments.

The problem of efficient DSL engineering is not new: substantial work has been done in the area the past few decades. Instead of creating a DSL from scratch, a DSL can be embedded into another host language [31]. Such embedded DSLs can be deep, meaning that a domain-specific program is translated into an internal data representation for further transformation and optimization, or it can be shallow where the DSL is encoded directly as part of the host language. Although there are several promising research results reported in the literature [3, 6, 9, 17, 29, 47, 52, 63], one of the main challenges with the embedded DSL approach is leaking abstractions: programming language abstractions and error messages from the host language are unintentionally exposed to the DSL end user.

A step further is to use complete DSL development frameworks, often referred to as language workbenches [22, 24]. Such frameworks [21, 23, 33, 35, 36, 56, 57] typically include
The vision of the proposed framework that is under development, called Miking (the "Meta Viking")¹, lies within this category of DSL language workbenches. However, in contrast to most of the available frameworks, which focus on DSLs for software, Miking targets complex domain-specific languages for mathematical modeling. In particular, the framework initially focuses on two DSL categories: (i) modeling languages for cyber-physical systems using differential-algebraic equations, difference equations, and timed state machines, and (ii) domain-specific differentiable probabilistic programming languages.

A language workbench for such DSLs puts extra requirements on (i) support for interactive modeling, (ii) reuse of existing language constructs and compilation strategies, and (iii) high-performance computation. Specifically, this paper discusses three key research areas within such a framework: Interactive Programmatic Modeling (Section 2), Sound Language Composition (Section 3), and Self-Learning Compilation (Section 4). Figure 1 gives an overview of the framework and how it is related to these three areas.

¹https://miking.org/ will include the framework when it has been released.

2 Interactive Programmatic Modeling

This section briefly describes the DSL domains and the main research challenges.

2.1 Mathematical Modeling Domains

The Miking framework is designed as a general-purpose language workbench. However, because generality easily leads to suboptimal solutions, the work initially focuses on two specific domains.

The first category of DSLs is modeling languages for cyber-physical systems (CPS). This includes a hierarchy of languages where instances of models (often referred to as components) can communicate with each other. These languages are typically timed, which means that continuous-time and discrete-time components must coexist and communicate with each other. Some of the existing domain-specific languages within this category are Modelica [41]—primarily used for modeling the dynamics of physical parts of a system, and Ptolemy II [15]—a software framework focusing on the mixture of different formalisms, such as discrete event, state machines, or synchronous data flow. There is also a large number of research DSLs within this category, such as Acumen [53], Zélus [7], Modelyze [9], Modia [18], and Hydra [29]. The novelty of our research DSL is the unique combination of...
acausal modeling as pioneered in Modelica, together with component-based mixture of computational formalisms, as advocated in Ptolemy II. The former makes use of hybrid differential-algebraic equations, whereas the latter is based on a composition semantics where individual components are orchestrated by a director, similar to a master algorithm in the functional mock-up interface (FMI) standard [8, 12].

The second category of mathematical DSLs is differentiable probabilistic programming languages (DPPL). This is a rather new research direction, where probabilistic programming constructs are combined with first-class language support for automatic differentiation [4, 5]. Probabilistic programming languages (PPLs) [10, 30, 40, 45, 64] have been around for many years, starting with languages for describing static Bayesian networks [28]. However, the PPL research area has recently received significant attention due to the development of new and more expressive universal PPLs.

DSLs within both these language categories may be seen as rather complex DSLs. The benefits of using a language workbench for these kinds of DSLs are to: (i) allow the development of hierarchies of DSLs, where for instance general PPLs can be specialized into domain-specific use cases, and (ii) to enable code reuse between DSLs. For instance, in one of our projects, the aim is to develop a specialized probabilistic DSL for computing phylogenetic trees. Such DSL should both reuse domain-specific optimizations [42] and be specialized for biologists with limited programming experience.

2.2 Interactive Programmatic Modeling

In the language categories described in the previous section, the end user regards the input programs as mathematical models. The term model, as used in this context, should not be confused with software models, such as UML models.

Although the domain users view the DSL programs as models, they are in fact programs. To emphasize this fact, we use the term programmatic models [43] to describe these DSL instances. As a consequence, the notation of programmatic models is textual, much due to the simplicity to express more complex models, compared to if the models were graphical. However, in many domains, such as the electrical or mechanical domains, visualization of the model is important to grasp the overall structure. In contrast to most available languages and tools for these categories, we envision an interactive user interface, where the input to the model is textual and different graphical views of the model are automatically updated using automatic layout algorithms, in the same spirit as discussed by Fuhrmann and von Hanxleden [27].

Another aspect is the interactive dynamic semantics and runtime output. Our aim is to explore the early ideas of illustrative [25] and example-centric [14] programming, which unfortunately have not been very influential in the mathematical modeling domain. The key idea is that the programs (programmatic models) and the output (simulation, inference, or verification results) coexist, much like how computational results and formulas coexist in spreadsheet programs. Such an approach of interactive programmatic modeling also relates to the concept of live programming [37, 55], which has been used in graphical environments [54], textual environments [32], and lately both for DSLs and modeling [59, 60]. An interesting research direction is also to combine such an approach with the record and replay debugging strategy, as advocated in the rr debugging tool [49].

2.3 Research Challenges

Key research challenges include, but are not limited to:

- Defining formal type systems, and performing static analysis and optimization of DPPLs, to achieve high-performance model inference.
- Development and encoding of formal semantics for heterogeneous CPS DSLs, including both timed runtime semantics, and static type systems.
- Performance optimization strategies to enable interactive real-time performance between model modifications and graphical view changes.

3 Sound Language Composition

This section discusses the problems and research challenges of introducing sound composition of language fragments.

3.1 Composition of Language Fragments

An important part of a framework for creating DSLs is its ability of extensibility. That is, to what extent is it possible to derive new DSLs from existing DSLs, without modifying the existing DSLs. In particular, in this work we advocate the possibility to construct new languages by composing small, unrelated language fragments. Ideally, domain experts with limited knowledge of programming language theory and compilers can create new complex sophisticated DSLs by only composing existing language fragments. Following the terminology by Erdweg et al. [20], we define two compositions that are relevant in our setting:

- **Language extension**, where $L_1 \triangleleft L_2$ is a new language formed by extending the base language $L_1$ with an extended language fragment $L_2$.
- **Language unification**, where $L_1 \triangleright L_2$ is the deep unification of the two languages $L_1$ and $L_2$, meaning that programs can be written consisting of terms from both languages, which can also interact with each other.

In the original classification [20], language unification also includes a glue code component, which makes the operator not necessarily symmetric. The aim of the Miking framework is to make unification both symmetric and associative. Thus, a glue code language $L_g$ can be part of the composition using a combination of the two operators: $(L_1 \triangleright L_2) \triangleleft L_g$.

Composability without static guarantees have been shown to work in practice at the syntactic level [44] and using complete language workbenches [16, 22, 61]. However, it is still
an open problem to be able to guarantee sound composition of language fragments, although recent progress has been made within the area of attribute grammars [34] or based on type-dependent syntactic extensions [39]. For instance, a desirable property of statically typed languages is type soundness, where “well-typed programs cannot go wrong”. Sound composition in regards to type soundness then means the following: Assume that two language fragments $L_1$ and $L_2$ have independently been proven to be type sound. Then a sound composition operator (of the meta language) would either compose $L_1$ and $L_2$ and return the composed language with the guarantee that the composed language is type sound, or it will reject and state that the composition is not safe.

3.2 Research Challenges

Research challenges include, but are not limited to:

- Ambiguity detection and mitigation of syntax composition in a sound and user friendly manner.
- Enabling scalable and efficient runtime systems within a methodology where languages are created from small language fragments. This includes both scalability in terms of composing sophisticated type systems and efficient compilation of composed languages.
- Defining a formal composition semantics, based on for instance System F$_{	ext{ωω}}$, that can be proven to be sound using Coq or Isabelle.

4 Self-Learning Compilation

The following section describes the main idea of self-learning compilation and the main research challenges.

4.1 Learning, Autotuning, and Optimization

During the last decades, various machine learning techniques for optimizing compilers have been extensively studied. Specifically, autotuning compilers focus on two major problems (i) selecting the best set of optimizations, and (ii) deciding the phase-ordering of the optimizations [2]. This research on autotuning, or optimizations such as polyhedral compilation [62], focuses on low-level compilation, typically on imperative code. SPIRAL [46] is another project that gives very good performance, especially on digital signal processing (DSP) transformations. Other notable efforts in this direction are compilers based on algorithmic skeletons [11], such as the Lift [51] intermediate language, SkeCL [50], and SkePU [13, 19]. These efforts are all based on parallel patterns. Another direction is to use partial evaluation as part of the framework [38], or to use a staging approach [48, 52].

In contrast to autotuning frameworks and parallel computation libraries, our view of self-learning compilation concerns automated high-level optimization and tuning (without user interaction), at the level where software developers would traditionally tune the programs themselves. Specifically, we have identified three main areas:

- Avoiding recomputations. The goal of the self-learning compiler is to automatically identify code that may be recomputed, and to insert code that mitigate recomputations. Example of such strategies can be to automatically insert partial evaluation of functions, or automatic strategies to perform memoization.
- Parallel computations. The goal is to automatically identify where parallelization is possible, and if it is beneficial. The former can be extremely hard in programs with side effects, whereas it is trivial in a pure functional setting. The latter suffers from combinatorial explosions, especially for context sensitive analysis.
- Selection of algorithms. The goal is to learn and automatically select the most efficient algorithms in certain contexts, by utilizing user defined annotations of possible algorithms. The performance metrics can for instance be time complexity, space complexity, or measured runtime behavior.

Such learning and optimization can be performed online during the execution of the program, offline by profiling benchmark programs before performing optimizations at compile time, or a combination of both. Online computations can learn from real data, but inherently lead to runtime overhead. Offline learning, on the other hand, does not suffer from runtime overhead, but can give suboptimal solutions.

4.2 Research Challenges

Some of the main research challenges are to:

- define the learning model, both for online and offline learning, which is representative for optimization.
- define strategies for collecting data, offline using representative benchmarks, or online with low overhead.
- combine static analysis and formal type systems to reason about effects and possibility for partial evaluation, algorithm selection, and parallelization.

5 Conclusions

This paper gives a brief overview of the vision of Miking, a proposed framework for constructing domain-specific modeling languages. The new research direction focuses on three aspects: (i) direct user feedback through an interactive programmatic modeling environment, (ii) sound composition of language fragments, and (iii) self-learning compilation. This work-in-progress project is currently at an early stage, and will be released as open source.

Acknowledgments

This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032) and by the Swedish Research Council, Vetenskapsrådet (#2018-04329). Special thanks to the anonymous reviewers and to Elias Castegren, Oscar Eriksson, Saranya Natarajan, Daniel Lundén, and Viktor Palmkvist for many useful comments.
References


