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:
David Broman. 2019. A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 6 pages. 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

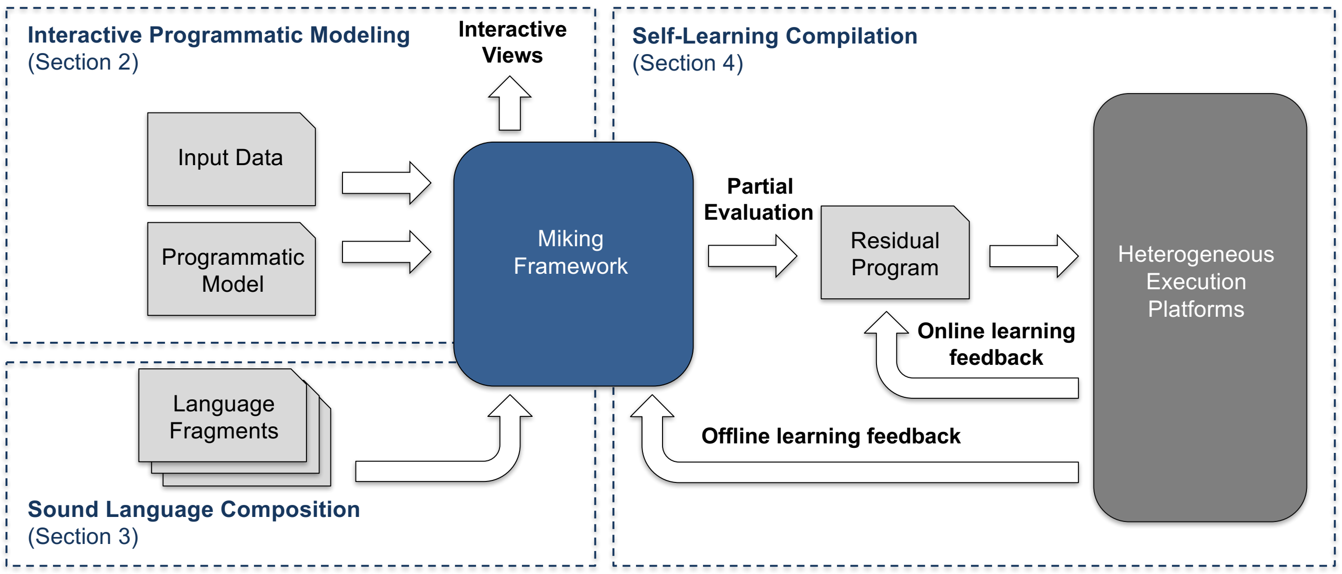

Figure 1. The figure gives an overview of the vision of Miking. The framework is divided into three parts. (i) The framework's approach to *interactive programmatic modeling* (Section 2) is depicted in the upper left part of the figure. Interactive views, such as graphical representations of the programmatic model and execution results that depend on specific input data are updated in separate interactive views. (ii) The lower left part of the figure (Section 3) illustrates *sound language composition*, where separate language fragments are composed and used when executing the programmatic models. (iii) The right side of the figure (Section 4) depicts the *self-learning compilation* approach of the framework. Composed language fragments and programmatic models are partially evaluated (specialized) into a residual program, which is executed on a heterogeneous platform. Profiling data are gathered online during program execution, or offline when executing specific benchmark programs. The profiled training data are then used by the self-learning compilation environment to improve execution performance.

DSL specific editors, compilers, test environments, debugging, and various syntactic and semantic services.

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) mathematical modeling languages for modeling 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.

[1]https://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 $L_1 \uplus L_2$ 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: $(L_1 \uplus L_2) \triangleleft L_g$.

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_ω, 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, SkelCL [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

[1] Peter J. Ashenden, Gregory D. Peterson, and Darrell A. Teegarden. 2002. *The System Designer's Guide to VHDL-AMS: Analog, Mixed-Signal, and Mixed-Technology Modeling.* Morgan Kaufmann Publishers, USA.

[2] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 96.

[3] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. 2008. Paradise: a two-stage DSL embedded in Haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP '08).* ACM, New York, NY, USA, 225–228.

[4] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research* 18 (2018), 1–43.

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6.

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming.* ACM Press, New York, USA, 174–184.

[7] Timothy Bourke and Marc Pouzet. 2013. Zélus: A synchronous language with ODEs. In *Proceedings of the 16th international conference on Hybrid systems: computation and control.* ACM, 113–118.

[8] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. 2013. Determinate Composition of FMUs for Co-Simulation. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2013).* IEEE.

[9] David Broman and Jeremy G. Siek. 2018. Gradually Typed Symbolic Expressions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '18).* ACM, New York, NY, USA, 15–29.

[10] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* 76, 1 (2017), 1–32.

[11] Murray I Cole. 1989. *Algorithmic skeletons: structured management of parallel computation.*

[12] Fabio Cremona, Marten Lohstroh, David Broman, Edward A Lee, Michael Masin, and Stavros Tripakis. 2017. Hybrid co-simulation: it's about time. *Software & Systems Modeling* (2017), 1–25.

[13] Usman Dastgeer, Johan Enmyren, and Christoph W Kessler. 2011. Autotuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering.* ACM, 25–32.

[14] Jonathan Edwards. 2004. Example centric programming. *ACM Sigplan Notices* 39, 12 (2004), 84–91.

[15] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (January 2003), 127–144.

[16] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system–modular extensible compiler construction. *Science of Computer Programming* 69, 1-3 (2007), 14–26.

[17] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming (ICFP '97).* ACM, New York, NY, USA, 263–273.

[18] Hilding Elmqvist, Toivo Henningsson, and Martin Otter. 2016. Systems modeling and programming in a unified environment based on Julia. In *International Symposium on Leveraging Applications of Formal Methods.* Springer, 198–217.

[19] Johan Enmyren and Christoph W Kessler. 2010. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications.* ACM, 5–14.

[20] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. 2012. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA).* ACM, 7.

[21] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) (OOPSLA '11).* ACM, 391–406.

[22] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[23] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (LIPIcs)*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128.

[24] Martin Fowler. 2005. Language workbenches: The killer-app for domain specific languages. Available from: https://martinfowler.com/articles/languageWorkbench.html. [Last accessed: June 28, 2019].

[25] Martin Fowler. 2009. Illustrative Programming. https://martinfowler.com/bliki/IllustrativeProgramming.html. [Last accessed: June 28, 2019].

[26] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 63–78.

[27] Hauke Fuhrmann and Reinhard von Hanxleden. 2010. On the Pragmatics of Model-Based Design. In *Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development (LNCS)*, Vol. 6028. Springer, 116–140.

[28] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* 43, 1 (1994), 169–177.

[29] George Giorgidze and Henrik Nilsson. 2008. Embedding a Functional Hybrid Modelling Language in Haskell. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages.*

[30] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'08).* AUAI Press, Arlington, Virginia, United States, 220–229.

[31] Paul Hudak. 1996. Building domain-specific embedded languages. *Comput. Surveys* (1996), 196.

[32] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, New York, NY, USA, 318–326.

[33] JetBrains. 2019. MPS: The Domain-Specific Language Creator by JetBrains. https://www.jetbrains.com/mps/. [Last accessed: June 28, 2019].

[34] Ted Kaminski and Eric Van Wyk. 2017. Ensuring non-interference of composable language extensions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE).* ACM, 163–174.

[35] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and

IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 444–463.

[36] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 168–177.

[37] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The road to live programming: insights from the practice. In *Proceedings in the 40th International Conference on Software Engineering (ICSE)*. IEEE, 1090–1101.

[38] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A partial evaluation framework for programming high-performance libraries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 119.

[39] Florian Lorenzen and Sebastian Erdweg. 2016. Sound Type-dependent Syntactic Language Extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 204–216.

[40] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099* (2014).

[41] Modelica Association 2017. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.4*. Modelica Association. Available from: http://www.modelica.org.

[42] Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *Proceedings of Machine Learning Research : International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR.

[43] Lawrence M Murray and Thomas B Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.

[44] Viktor Palmkvist and David Broman. 2019. Creating Domain-Specific Languages by Composing Syntactical Constructs. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*. Springer, 187–203.

[45] Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report* 137 (2009), 96.

[46] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE* 93, 2 (2005), 232–275.

[47] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* 25, 1 (2012), 165–207.

[48] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136.

[49] rr development team. 2019. rr: lightweight recording & deterministic debugging. https://rr-project.org. [Last accessed: June 28, 2019].

[50] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL-a portable skeleton library for high-level gpu programming. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 1176–1182.

[51] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85.

[52] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s, Article 134 (April 2014), 134:1–134:25 pages.

[53] Walid Taha, Adam Duracz, Yingfu Zeng, Kevin Atkinson, Ferenc A Bartha, Paul Brauner, Jan Duracz, Fei Xu, Robert Cartwright, Michal Konečný, et al. 2015. Acumen: An open-source testbed for cyber-physical systems research. In *International Internet of Things Summit*. Springer, 118–130.

[54] Steven L Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139.

[55] Steven L Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press, 31–34.

[56] Enso team. 2019. Ensō: A self-describing DSL workbench. http://www.enso-lang.org/index.shtml. [Last accessed: June 28, 2019].

[57] Juha-Pekka Tolvanen and Matti Rossi. 2003. MetaEdit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 92–93.

[58] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: an annotated bibliography. 35, 6 (2000), 26–36.

[59] Riemer van Rozen and Tijs van der Storm. 2019. Toward live domain-specific languages. *Software & Systems Modeling* 18, 1 (2019), 195–212.

[60] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2019. A Multi-Paradigm Modelling approach to live modelling. *Software & Systems Modeling* 18, 5 (2019), 2821–2842.

[61] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1-2 (2010), 39–54.

[62] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[63] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, USA, 242–252.

[64] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*. 1024–1032.