Sprat: Hierarchies of Domain-Specific Languages for Marine Ecosystem Simulation Engineering

Arne Johanson, and Wilhelm Hasselbring
Software Engineering Group, Department of Computer Science, Kiel University, Kiel, Germany,
and Helmholtz Research School for Ocean System Science and Technology (HOSST),
GEOMAR – Helmholtz Centre for Ocean Research, Kiel, Germany
Email: (arj, wha)@informatik.uni-kiel.de

Keywords: Domain-specific languages, DSL, model-driven software engineering

Abstract

Scientific software is becoming more complex and its development nowadays is often an interdisciplinary process in which usually scientists – most of them without training in software engineering – implement the software themselves. To help them achieve a good code quality, we propose to employ a process called Sprat based on the concept of hierarchies of domain-specific languages (DSLs). In such a hierarchy, every DSL constitutes an implementation platform for the DSL directly above it. Each role in the development process (i.e., a scientist from a specific discipline) implements a layer of the software in a DSL belonging to the hierarchy. Therefore, the scientists only deal with abstractions they are familiar with from their respective domain and a clear separation of components is attained. To evaluate the Sprat approach, we present its application to the development of the Sprat simulation – a marine ecosystem model for long-term fish stock prediction.

1. INTRODUCTION

With *in silico* experiments becoming more important in science [8], the complexity and lifespan of scientific software is increasing as well as the need for results of scientific simulations to be reproducible and verifiable. To meet these challenges efficiently, the developers of scientific software – who are usually the computational scientists themselves – must produce a maintainable and testable code base. But because of the "wide chasm" [9] between the disciplines of scientific computing and software engineering, most of these scientists received no training in software engineering that would have taught them established tools and best practices to achieve these goals. This problem, however, cannot only be attributed to a knowledge gap among computational scientists but can also be seen as resulting from the fact that most research in software engineering has been focused on the development of business and embedded software. Because of this and because of the different role that software plays in the scientific community [6], software engineering tools and best practices cannot simply be transferred to computational science but have to be adapted to the specific domain.

In response to these challenges, we propose an approach called Sprat that is based on the concept of hierarchies of domain specific languages (DSLs). Employing this concept, allows scientists from different fields to collaborate on the implementation of scientific software, utilizing the abstractions they are familiar with from their domain. Due to the high level of abstraction of such DSLs, they allow solutions to be expressed much more concisely than with general purpose programming languages. Thus, if the languages are well-designed, they encourage and simplify the process of writing code that is easy to maintain and to test.

In order for such an approach to be accepted by the scientific computing community, it has to be ensured that no "accidental complexity" [24] is introduced along with it. In practice, this means that the effort to become proficient in a DSL from the hierarchy must be very small for the targeted domain-experts. Besides, tool support for the DSLs has to be provided. And finally, the increased level of abstraction should not compromise the run-time performance of programs (a requirement that is especially important for the high-performance computing (HPC) community) and should introduce as few dependencies on libraries, etc. as possible.

In the rest of the paper we characterize Sprat and the concept of hierarchically organized DSLs in greater detail by describing how to apply it to the implementation of a spatially explicit partial differential equation (PDE)-based ecosystem model for long-term fish stock prediction. Section 2 presents the foundations of our hierarchical DSL approach. Its application to the named ecosystem model is discussed in Section 3. Finally, we describe related work (Section 4) as well as our conclusions and further research questions (Section 5).

2. HIERARCHIES OF DSLS

The fundamental idea of model-driven software development (MDSD) is to generate as many artifacts of a software project as possible from an abstract representation – a model – of the solution to be implemented. This model can either be described graphically or textually by a so-called domain-specific language (DSL).

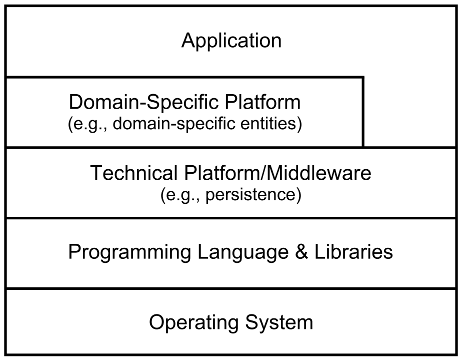
Application
Domain-Specific Platform
(e.g., domain-specific entities)
Technical Platform/Middleware
(e.g., persistence)
Programming Language & Libraries
Operating System

Figure 1. A DSL build upon a semantically rich platform. Figure adapted from [18].

In this paper, we follow Stahl and Völter [18] who take a rather pragmatic look at MDSD by focusing less on model-to-model transformations but more on code-generation from a model. Thus, they add to the model as the first cornerstone of MDSD the concept of a "semantically rich," domain-specific platform as a second key element (cf. Fig. 1). As such a platform already incorporates reusable components from the specific domain, model-to-code transformations are made easier.

We extend this idea of transformations between layers that are semantically oriented to each other by introducing several of these layers organized in a hierarchical fashion. Each of these layers is itself a DSL that is implemented by the DSL directly beneath it in the hierarchy. In this way, the implementing DSL becomes a semantically rich platform for the implemented DSL (cf. Fig. 3, which shows this relation for our evaluation example).

Unlike the suggested vertical hierarchy, multiple DSLs are usually only organized in a horizontal structure. This means, that they divide the targeted domain in sub-domains and share some common aspect of the domain-metamodel which allows them to interact with each other (cf. Fig. 2).

In such a horizontal organization of multiple DSLs, *different aspects* of a system can be expressed in a specifically tailored language. With our approach of DSL hierarchies, *different systems* that build upon each other can all be implemented by different domain experts. In the context of scientific software development, this means that scientists from different areas are enabled to collaborate on software projects while only working with abstractions they are familiar with from their respective domain. Due to the high expressiveness of a well-designed DSL, the code of an implemented solution that uses this language can be very concise. This simplifies writing code that is easy to maintain and to evolve because it is almost self-documenting.

The modular approach of hierarchically organized DSLs also facilitates the reuse of code because layers of the hierarchy can be interchanged or ported with low effort. This is because all generators in the hierarchy are quite simple as each

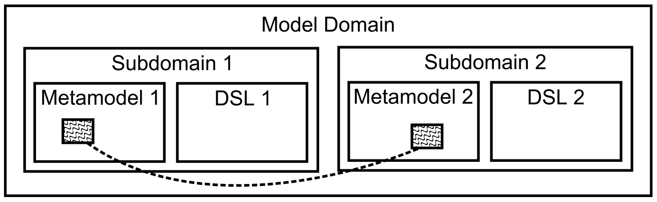

Figure 2. Typically, domains (and thus systems) are split in a horizontal fashion. Figure adapted from [18].

upper layer can rely on reusable domain-specific components already implemented in the lower layer.

As domains evolve over time, the DSLs of the hierarchy must be adapted to capture new concepts. While most certainly a software engineer will have to collaborate with domain experts to create the initial version of the DSLs, the aforementioned relative simplicity of the generators potentially allows the DSL users to maintain the languages themselves.

3. EVALUATION OF THE SPRAT APPROACH

In order to evaluate the Sprat approach of hierarchically organized DSLs for scientific application development and to see how its benefits – as presented in Section 2 – can be obtained in practice, we apply the process to the implementation of the *Sprat model*.

The Sprat model is a PDE-based ecosystem model for long-term fish stock prediction. To approximate the solution of its PDEs that are defined on an up to four-dimensional set, they are discretized using a flux-corrected finite-element method (FEM) [10].

In this context, we introduce three DSLs that are organized in a hierarchical fashion (Fig. 3). The first one (*Sprat PDE solver DSL*) allows the efficient implementation of our special-purpose mesh-based PDE-solver algorithm. It is also used to express the functional relationships comprised in the Sprat model. The model and its solver implemented with this language constitute the platform for the second language (*Sprat ecosystem DSL*). It is designed to specify the parameters of the simulation, such as the involved fish species, as well as the parameters to be aggregated and recorded during the simulation. The parameterized simulations described by the ecosystem DSL again establish the platform for the third language in the hierarchy (*Sprat deployment DSL*). Its purpose is to describe how to map a simulation to a specific (distributed) execution environment. These three DSLs together with the model itself constitute the *Sprat simulation*.

As indicated in Figure 3, the simulation implemented with Sprat ecosystem DSL does not have to be realized using the Sprat PDE solver DSL but can work with any other existing simulation tool. In fact, it does not even have to be re-

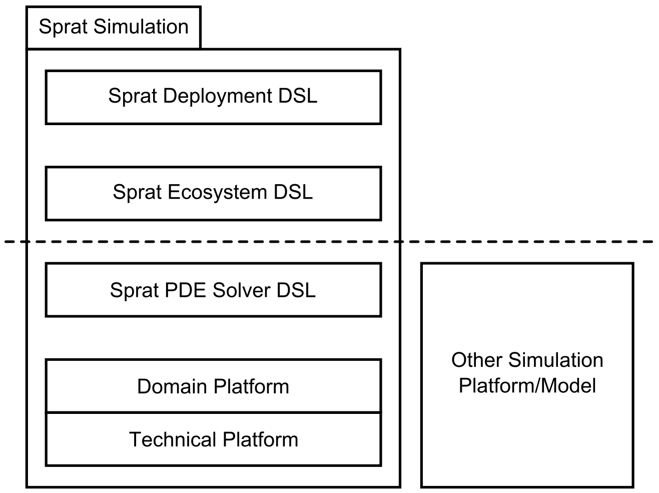

Figure 3. Hierarchical DSL structure of the Sprat simulation.

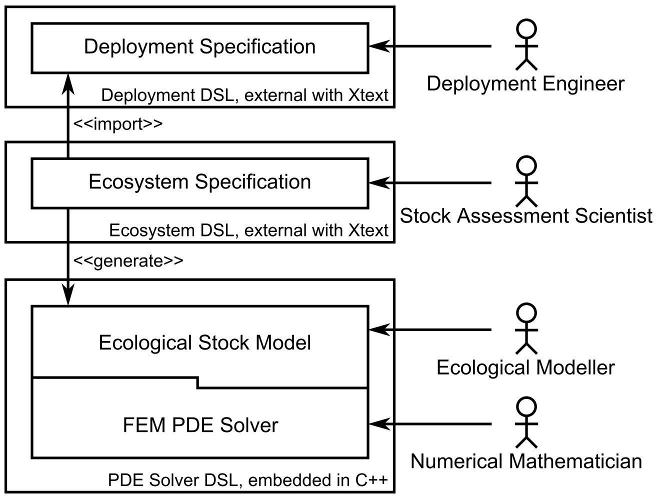

Figure 4. Artifacts and roles associated with the different DSLs of the Sprat simulation.

alized by the Sprat ecosystem model at all: as a simulation implemented with the Sprat ecosystem DSL is stated solely in terms of fisheries ecology models, it can be mapped to any other fish stock model that is or can be formulated using these abstractions. The only thing that has to be adapted to enable such an exchange is the light-weight generator. This allows the scientist to compare the results of different models for the exact same input description by merely switching to another generator. And the Sprat PDE solver DSL in return can, of course, also be used to implement other simulations that are PDE-based.

Figure 4 presents the Sprat DSL hierarchy from the perspective of the different roles involved in the implementation of the simulation as well as the artifacts created by those roles.

The use of different DSLs by different roles establishes a clear separation of concerns and thus a clear modularization of the implemented software. In the case of Sprat there is one noteworthy exception to this as the role of the ecological modeller and the role of the numerical mathematician implement their artifacts in a common DSL and the artifacts produced by them overlap. While it is clear that they can share a common language (both express their results in “mathematical formulas”) it would surely be desirable for their artifacts not to be intertwined with each other. This overlap is the result of a trade-off between modularization and performance: in order to be run-time efficient, the model implementation has to make use of some implementation details of the solver. This can be tolerated because the important separation between the model and the solver can still be expressed quite clearly.

The following subsections discuss the design and technical implementation of the Sprat PDE solver DSL and the ecosystem DSL.

3.1. Sprat PDE Solver DSL

The Sprat PDE solver DSL is designed to facilitate the implementation of numerical algorithms for mesh-based PDEs with a focus on finite-element methods especially for biological applications. In such applications, the unknown functions often do not only depend on spatial coordinates but also on other continuous quantities associated with every spatial point (e.g., the number of some individuals can be modeled to depend on the spatial position as well as the size of individuals). Because of these additional *internal coordinates* [16], the PDE solver DSL is designed to work with domains of arbitrary dimension. Additionally, PDEs from a biological modeling context can exhibit non-local behavior (e.g., individuals do not only take into account their *immediate* neighborhood). Therefore, special emphasis is given to data-structures for the efficient representation of such interactions.

The DSL is embedded into C++ and relies on a combination of operator overloading and template metaprogramming to convey the impression of an independent language [3]. We embed the PDE solver DSL into a general-purpose language as the numerical mathematician must have such a language at hand because of the constant need to implement problem-specific data-structures. In this way, we also obtain full tool support for the quite complex language.

C++ as a host language was mainly chosen for user-acceptance reasons: it can be seen as an extension to C, which is well-established in the domain of scientific computing. Thus, the scientific programmer most likely does not have to learn a new language. In addition to that, C++ is also suitable because of its template metaprogramming capabilities that support the technique of expression templates [22, 1]. The latter enable lazy evaluation of operator expressions and thus eliminate the general need for temporary variables during this process. Thereby, the execution time of such DSL operations can be on par with a much less expressive hand-coded FORTRAN implementation of the same functionality [23]. Sprat's PDE solver DSL implements expression templates by using the Boost Proto library [13].

In the following paragraphs we describe the central additions that the PDE solver DSL introduces to C++.

Matrix-Vector Arithmetic Matrix-vector expressions can be written in the way known from linear algebra. The evaluation of such expressions must not require temporaries unless the user explicitly asks for this. In this way, the user remains in full control of memory allocation, which is especially important in the HPC context. The compact notation of such arithmetic expressions encourages defensive programming techniques as this notation simplifies the task of checking assertions to numerical quantities.

Vectors For working with vectors, Sprat's PDE solver DSL offers views only on selected elements of a vector that can themselves be used just like a vector. Additionally, there are reduce functions using lambda expressions.

Matrices/Graphs The PDE solver DSL features sparse matrix types as the operators occurring in FEM applications usually are of this type. These sparse operators can be addressed both as matrices and as graphs. This is advantageous because in biological applications there often is the need to perform non-standard operations on more or less local stencils, such as finding the maximum of a discretized function in a certain surrounding of every grid point. The DSL enables to express such an operation using the same notation as with matrix-vector expressions by specifying custom behavior for operator application:

```
Graph M;  Vector a, b;
Vector localMax = ( M*(a+b),
                    reduceLocalMax );
double reduceLocalMax(...) { ... }
```

Meshes There are data types for unstructured and (partially) structured meshes for domains of arbitrary dimension. Additionally, the language features **sets** of the composing structures of the mesh (elements, edges, vertices, hypersurfaces, hypersurfaces of hypersurfaces etc.). For the function of those sets refer to the next item.

Iterations Over Sets Manipulating indices for iterating over collections of objects is a major source of errors in numerical programs. To eliminate this source and to make clearer what an iteration represents (quick, what does the infamous iteration index `i` stand for in this specific loop?), Sprat's PDE solver DSL introduces for-loops over sets (for the concept cf. [5]). These loops can be parallelized using OpenMP [4] as it is possible with any other for-loop:

```
#pragma omp parallel for
for(auto e : Elements(mesh)) { ... }
```

SPMD Abstractions PDE-based simulations are typically parallelized using the single program multiple data (SPMD) technique. The PDE solver DSL offers some constructs to reduce the overhead of such an implementation. Namely, there is a parallel execution environment that abstracts away the concrete implementation of data exchange. Furthermore, the language features the concepts of mesh partitions and overlapping regions. By using these abstractions, also developers not experienced in large-scale parallel programming are likely to achieve performance levels otherwise hard to reach for them (regarding the importance of experience and effort for parallel programming for clusters cf. [20]).

It is especially the natural syntax of the matrix-vector arithmetic and the concept of iterations over sets which justifies calling the PDE Solver DSL an embedded *language* rather than a *library*.

3.2. Sprat Ecosystem DSL

The Sprat ecosystem DSL is intended to let ecologists specify the parameters of an ecosystem simulation and the output variables to be recorded during a simulation run. The language abstracts away the internals of a specific simulation package and removes the need for the biologist to bother with different output data formats as well as the often complicated APIs of the associated software libraries.

To achieve a language that is easy to learn, full control over the syntax is necessary. That is why the ecosystem DSL is implemented as an external DSL using Xtext [7]. Xtext is based on the Eclipse Modelling Framework [19] and lets the user specify a grammar in a notation close to the Extended-Backus-Naur-Form. From the latter a parser, a code generator stub, and an Eclipse IDE with syntax highlighting as well as an auto complete feature are generated. Besides a more natural syntax for the DSL, complete tool support is the second cornerstone for working with such a language efficiently.

The metamodel of Sprat's ecosystem DSL is organized around four different **entities** that can each have specific **properties**. The top-level entities are `Ecosystem`, `Input`, `Output`, and `Species`. Multiple instances can appear only of the latter. The properties are attribute-value pairs that are associated with an entity and potentially have a unit assigned to them.

```
Species:"Herring" {
  MaxAge: 25 years
  YolkSackDepletedAfter: 21 days @ 10 °C
}
```

In the `Output` entity, the notion of properties is supplemented by the `record` keyword and arithmetic expressions that can involve internal functions. Thereby, the desired output parameters of the simulation can be specified, which eliminates the need for a distinct post-processing step.

```
Output {
  record@everyTimestep "Some Quantity":
    biomass(species="Herring",
      weight=500 g to 1 kg) *
    fecundity(species="Herring")
}
```

To go into more detail about the language design would be beyond the scope of this paper.

4. RELATED WORK

In this section, we discuss work related both to our concept of hierarchical DSLs as well as to the PDE solver DSL that we employ in the implementation of our evaluation scenario.

4.1. DSL Hierarchies

In the context of his pattern language for DSLs, Spinellis [17] uses the term *DSL hierarchy* for the hierarchical inheritance structure that results from extending a base language with special syntax to form a DSL. In contrast to this, our definition of DSL hierarchies is broader. Even DSLs implemented with completely different technologies can belong to the same hierarchy as long as each of them forms a platform for the level above it in the sense described in Section 2.

Neighbors [12] presents an approach to reusable software components that makes use of multiple DSLs to allow a system designer to refine domain concepts into other domains. Although the notion of a hierarchical organization of these languages is present, the author explicitly states that they do not form a strict hierarchy. Apart from being strictly hierarchical, our approach differs from the one presented in [12] insofar as ours strongly emphasizes the separation of different roles of scientific software developers in interdisciplinary projects.

Preschern, Leitner, and Kreiner [15] as well as Prähofer and Hurnaus [14] highlight the importance of hierarchical concepts for DSLs in the context of automation systems. But instead of introducing multiple DSLs that are arranged in a hierarchical fashion, they suggest a single DSL that incorporates the concept of hierarchically nested models [15] or hierarchical components [14] respectively.

4.2. Sprat PDE Solver DSL

Blitz++ [21], Eigen[1], and Armadillo[2] are C++ scientific computing libraries that provide expression templates for matrix-vector arithmetic. They, however, are more general than Sprat's PDE solver DSL in that they are not specifically tailored to mesh-based PDE algorithms and thus lack some important domain concepts for this purpose (especially for easy handling of the geometry). Additionally, these libraries focus on dense and not so much on sparse matrices.

FEniCS [11] provides a DSL (Finite Element Form Language [2]) for specifying FEM discretizations and variational forms. The level of abstraction of such a language is higher than with our DSL and targets the FEM practitioner rather than the algorithm developer. Additionally, the FEniCS framework is limited to three-dimensional problems as are most similar tools (cf. the related work section of [2]).

Liszt [5] aims at the same level of absctraction as does the Sprat PDE solver DSL but focuses on automatic parallelization rather than parallelization through high-level annotations. Algorithms implemented with it are limited to three dimensions as well.

5. CONCLUSION AND FUTURE WORK

Our approach Sprat of hierarchically organized DSLs promises to facilitate the cooperation of scientific software developers from different fields and, at the same time, to improve the code quality of such projects. The layer of abstraction of each DSL matches the expertise of a discipline involved in the project. As every language constitutes a semantically rich platform for the supraordinate DSL in the hierarchy, the effort to implement model-to-code transformations is low. The concise syntax of each language simplifies the task of writing clear code and thereby encourages the creation of a more maintainable code base that is easier to test. To demonstrate how this approach helps to adapt proven best practices of software engineering to the context of scientific computing and scientific software development, we described its application to the implementation of a PDE-based ecosystem model.

Future work includes:

- With Sprat's ecosystem DSL we want to be able to specify biological parameters of a marine ecosystem and evaluate the predicted evolution of this system not only using the Sprat model itself but different ecosystem models to compare the results. Because the mathematical formulations of these models differ from the one of the Sprat model, the question arises how to map a simulation description given in the ecosystem DSL to those different models without introducing redundant information into the description. To tackle this challenge, we plan to collaborate with the user community of other models in order to evaluate the potential for employing the Sprat ecosystem DSL in connection with the respective model.

- Concerning the Sprat deployment DSL it is to be decided

[1]http://eigen.tuxfamily.org

[2]http://arma.sourceforge.net

which concrete syntax and which runtime technology to use for the language. To this end, we will assess technologies with similar function, such as deployment descriptors in the context of Java.

- Furthermore, it could prove valuable to provide the Sprat simulation as a (web) service. In this way, a simulation can be scheduled merely by providing a description file written in Sprat's deployment DSL and the accompanying input data.

REFERENCES

[1] David Abrahams and Aleksey Gurtovoy. 2004. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Addison Wesley.

[2] M. S. Alnæs. 2012. "UFL: a Finite Element Form Language". In: *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Vol. 84. LNCSE. Springer. Chap. 17.

[3] Krzysztof Czarnecki et al. 2004. "DSL implementation in MetaOCaml, Template Haskell, and C++". In: *Domain-Specific Program Generation*. Vol. 3016. LNCS. Springer, pp. 51–72.

[4] Leonardo Dagum and Ramesh Menon. 1998. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science & Engineering, IEEE* 5.1, pp. 46–55.

[5] Z. DeVito et al. 2011. "Liszt: a domain specific language for building portable mesh-based PDE solvers". In: *Proceedings SC11*. ACM.

[6] S.M. Easterbrook and T.C. Johns. 2009. "Engineering the software for understanding climate change". In: *Computing in science & engineering* 11.6, pp. 65–74.

[7] Sven Efftinge et al. 2012. "Xbase: implementing domain-specific languages for Java". In: *Proceedings GPCE'12*. ACM, pp. 112–121.

[8] Jim Gray. 2009. "eScience: A Transformed Scientific Method". In: *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Ed. by Tony Hey, Stewart Tansley, and Kristin Tolle. Microsoft Research, pp. XVII–XXXI.

[9] J.E. Hannay et al. 2009. "How do scientists develop and use scientific software?" In: *Proceedings SECSE'09*. IEEE, pp. 1–8.

[10] Dimitri Kuzmin, Rainald Löhner, and Stefan Turek. 2012. *Flux-Corrected Transport: Principles, Algorithms, and Applications. Scientific Computation*. 2nd ed. Scientific Computation. Springer.

[11] A. Logg, K.A. Mardal, and G. Wells. 2012. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Vol. 84. LNCSE. Springer.

[12] James M. Neighbors. 1984. "The Draco approach to constructing software from reusable components". In: *Software Engineering, IEEE Transactions on* 5, pp. 564–574.

[13] Eric Niebler. 2007. "Proto: A compiler construction toolkit for DSELs". In: *Proceedings LCSD'07*. ACM, pp. 42–51.

[14] Herbert Prähofer and Dominik Hurnaus. 2010. "MONACO – A domain-specific language supporting hierarchical abstraction and verification of reactive control programs". In: *Proceedings INDIN 2010*. IEEE, pp. 908–914.

[15] Christopher Preschern, Andrea Leitner, and Christian Kreiner. 2012. "Domain specific language architecture for automation systems: an industrial case study". In: *Proceedings ECMFA'12*.

[16] Doraiswami Ramkrishna. 2000. *Population Balances: Theory and Applications to Particulate Systems in Engineering*. Academic Press.

[17] Diomidis Spinellis. 2001. "Notable design patterns for domain-specific languages". In: *Journal of Systems and Software* 56.1, pp. 91–99.

[18] Thomas Stahl and Markus Völter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

[19] D. Steinberg et al. 2008. *EMF: Eclipse Modeling Framework*. 2nd ed. Addison-Wesley Professional.

[20] Michael L Van De Vanter et al. 2009. *Productive petascale computing: requirements, hardware, and software*. Tech. rep. Sun Microsystems, Inc.

[21] Todd L. Veldhuizen. 2000. "Blitz++: The library that thinks it is a compiler". In: *Advances in Software tools for scientific computing*. Springer, pp. 57–87.

[22] Todd L. Veldhuizen. 1995. "Expression templates". In: *C++ Report* 7.5, pp. 26–31.

[23] Todd L. Veldhuizen and M. Ed. Jernigan. 1997. "Will C++ be faster than Fortran?" In: *Scientific Computing in Object-Oriented Parallel Environments*. Springer, pp. 49–56.

[24] G.V. Wilson. 2006. "Where's the real bottleneck in scientific computing?" In: *American Scientist* 94.1, pp. 5–6.