Preprint of: Arne N. Johanson, Wilhelm Hasselbring, Andreas Oschlies, Boris Worm:

View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by OceanRep

Edited by Jeffrey C. Carver, Neil P. Chue Hong, George K. Thiruvathukal. Taylor & Francis Group, CRC Press 2016 Pages 175–200. Print ISBN: 978-1-4987-4385-3
eBook ISBN: 978-1-4987-4386-0

Chapter 8

Evaluating Hierarchical Domain-Specific Languages for Computational Science: Applying the Sprat Approach to a Marine Ecosystem Model

Arne N. Johanson, Wilhelm Hasselbring, Andreas Oschlies, and Boris Worm

8.1 Motivation 176
8.2 Adapting Domain-Specific Engineering Approaches for Computational Science 177
8.3 The Sprat Approach: Hierarchies of Domain-Specific Languages 179
  8.3.1 The Architecture of Scientific Simulation Software 179
  8.3.2 Hierarchies of Domain-Specific Languages 181
    8.3.2.1 Foundations of DSL Hierarchies 182
    8.3.2.2 An Example Hierarchy 183
  8.3.3 Applying the Sprat Approach 185
    8.3.3.1 Separating Concerns 185
    8.3.3.2 Determining Suitable DSLs 186
    8.3.3.3 Development and Maintenance 188
  8.3.4 Preventing Accidental Complexity 189
8.4 Case Study: Applying Sprat to the Engineering of a Coupled Marine Ecosystem Model 190
  8.4.1 The Sprat Marine Ecosystem Model 190
  8.4.2 The Sprat PDE DSL 191
  8.4.3 The Sprat Ecosystem DSL 191
  8.4.4 The Ansible Playbook DSL 192
8.5 Case Study Evaluation 193
  8.5.1 Data Collection 193
  8.5.2 Analysis Procedure 195
  8.5.3 Results from the Expert Interviews 195
    8.5.3.1 Learning Material for DSLs 195
    8.5.3.2 Concrete Syntax: Prescribed vs. Flexible Program Structure 196

8.5.3.3 Internal vs. External Implementation 196
8.6 Conclusions and Lessons Learned 198

In this chapter, we present a Model-Driven Software Engineering (MDSE) approach called *Sprat*, which adapts traditional software engineering practices in order to employ them in computational science. The approach is based on the hierarchical integration of so-called Domain-Specific Languages (DSLs) to facilitate the collaboration of scientists from different disciplines in the development of complex simulation software. We describe how multiple DSLs can be integrated to achieve a clear separation of concerns among the disciplines and how to apply Sprat during the different phases of the software life cycle.

To evaluate our approach, we discuss results from a case study in which Sprat has been utilized for the implementation of a coupled marine ecosystem model for spatially-explicit fish stock prediction. We report on the DSLs developed for this case study, how scientists benefit from them, and on lessons learned. In particular, we analyze the results from expert interviews conducted with both scientists and professional DSL developers.

The remainder of this chapter is structured as follows: in Section 8.1, we motivate our research and point out a communication gap between software engineering and computational science. This gap makes it necessary to *adapt* software engineering techniques and methods for them to be adopted by scientists. Section 8.2 explains why Model-Driven Software Engineering (MDSE) approaches and Domain-Specific Languages (DSLs) are good starting points for such adaptations. In Section 8.3, we introduce our Sprat Approach and describe the case study we use for its evaluation in Section 8.4. We report on results from this case study in Section 8.5. Conclusions and lessons learned are given in Section 8.6.

8.1 Motivation

When software engineers started to examine the software development practice in computational science, they noticed a “wide chasm” [19] between how these two disciplines view software development. Faulk et al. [16] describe this chasm between the two subjects using an allegory which depicts computational science as an isolated island that has been colonized but then was left abandoned for decades:

> “Returning visitors (software engineers) find the inhabitants (scientific programmers) apparently speaking the same language, but communication—and thus collaboration—is nearly impossible; the technologies, culture, and language semantics themselves have evolved and adapted to circumstances unknown to the original colonizers.”

The fact that these two cultures are "separated by a common language" created a communication gap that inhibits knowledge transfer between them. As a result, modern software engineering practices are rarely employed in computational science.

So far, the most promising attempt to bridge the gap between computational science and software engineering seems to be education via workshop-based training programs focusing on Ph.D. students, such as the ones organized by Wilson [41] and Messina [29]. While the education approach does address the skill gap that is central to the "software chasm," education will not suffice alone: just exposing scientists to software engineering methods will not be enough because these methods often fail to consider the specific characteristics and constraints of scientific software development—i.e., the functioning of these methods is based on (often implicit) assumptions that are violated in the computational science context [20, 11]. We therefore conclude that—complementary to the education approach—we have to select suitable software engineering techniques and *adapt* them specifically to the needs of computational scientists.

8.2 Adapting Domain-Specific Engineering Approaches for Computational Science

Scientists are only "accidentally" involved in software development: ultimately, their goal is *not* to create software but to obtain novel scientific results. At the same time, however, they are very concerned about having full control over their applications and how these actually compute their results, which is why many prefer "older" programming languages with a relatively low level of abstraction from the underlying hardware (cf. [16, 33, 6, 11]).

Among the techniques and tools that software engineering has to offer, Model-Driven Software Engineering (MDSE) [38, 7] and especially Domain-Specific Languages (DSLs) [17] are promising starting points for addressing the needs of computational scientists.

MDSE uses *models* expressed in *modeling languages* as the primary artifact in every stage of the software life cycle (implying that models are also implementation artifacts). *Transformations* are employed to map a source model to a target artifact which can either be a model again (*model-to-model* transformation) or an arbitrary textual artifact like source code (*model-to-text* transformation).

The modeling languages utilized in the context of MDSE are so-called *Domain-Specific Languages (DSLs)*. Like General-Purpose Languages (GPLs), such as C or Java, DSLs are programming languages. However, unlike GPLs, which are designed to be able to implement any program that can be com-

puted with a Turing machine, DSLs limit their expressiveness to a particular application domain. By featuring high-level domain concepts that enable to model phenomena at the abstraction level of the domain and by providing a notation close to the target domain, DSLs can be very concise. The syntax of a DSL can be *textual* or *graphical* and DSL programs can be executed either by means of *interpretation* or through *generation* of source code in existing GPLs. A popular example of a textual DSL are regular expressions, which target the domain of text pattern matching and allow to model search patterns independently from any concrete matching engine implementation.

As with any other formal language, a DSL is defined by its *concrete* and *abstract syntax* as well as its *static* and *execution semantics*. While the concrete syntax defines the textual or graphical notation elements with which users of the DSL can express models, the abstract syntax of a DSL determines the entities of which concrete models can be comprised. These abstract model entities (abstract syntax) together with the constraints regarding their relationships (static semantics) can again be expressed as a model of all possible models of the DSL, which is therefore called the *meta-model* of the DSL.

Since DSLs are designed to express solutions at the abstraction level of the domain, they allow the scientists to care about what matters most to them: doing science without having to deal with technical, implementation-specific details. While they use high-level domain abstractions, they still stay in full control over their development process as it is them who directly implement their solutions in formal and executable (e.g., through generation) programming languages. Additionally, generation from a formal language into a low-level GPL permits to examine the generated code to trace what is actually computed.

DSLs can also help to reconcile the conflicting quality requirements of performance on the one hand and portability and maintainability on the other hand that are responsible for many of the difficulties experienced in scientific software development (cf. [11]). DSL source code is maintainable because it is often pre-structured and much easier to read than GPL code, which makes it almost self-documenting. This almost self-documenting nature of DSL source code and the fact that it can rely on an—ideally—well-tested generator for program translation ensure the reliability of scientific results based on the output of the software. Portability of DSL code is achieved by just replacing the generator for the language with one that targets another hardware platform. With DSLs, the high abstraction level does not have to result in performance losses because the domain-specificity first of all enables to apply—at compile time—domain-specific optimizations and greatly simplifies automatic parallelization.

In the way described above, DSLs integrated into a custom MDSE approach could help to improve the productivity of computational scientists and the quality of their software. A first indicator that supports this hypothesis can be found in the survey report of Prabhu et al. [31], who find that those scientists who program with DSLs "report higher productivity and satisfac-

tion compared to scientists who primarily use general purpose, numerical, or scripting languages."

8.3 The Sprat Approach: Hierarchies of Domain-Specific Languages

In this section, we introduce *Sprat*, which is a MDSE approach that aims at enabling scientists from different disciplines to efficiently collaborate on implementing well-engineered simulation software without the need for extensive software engineering training. Its underlying idea is to provide a Domain-Specific Language (DSL) for each (sub-)discipline that is involved in the development project and to integrate these modeling languages in a hierarchical fashion. The hierarchical structure of the DSL integration is enabled by the typical architecture of scientific (simulation) software, which we discuss before introducing Sprat itself.

8.3.1 The Architecture of Scientific Simulation Software

Typical scientific simulation software can be implemented using the multi-layered software architecture pattern [9]. A software system conforms to this pattern if its components can be partitioned into a hierarchy of layers in which each layer corresponds to a particular level of abstraction of the system. In addition to that, every layer has to be implemented using only the abstractions of lower layers but never using abstractions from higher ones. Popular examples of the application of the layers pattern are networking protocols, which introduce layered levels of abstraction ranging from low-level bit transmission to high-level application logic.

We argue that the general structure of typical simulation software lends itself to the layers pattern and clarify this with an example: generally speaking, scientific simulation software employs algorithms to analyze scientific models—i.e., mathematical abstractions of the real world—by means of computation. The scientific models can be formalized using different mathematical frameworks, such as differential equations or agent-based models [18]. Based on the respective mathematical framework and on the aspects that the model is supposed to be examined for, a suitable analysis algorithm is chosen.

For example, an ocean model would usually be based on the physical laws of fluid dynamics which are formulated as Partial Differential Equations (PDEs). An implementation of this model would employ a suitable PDE solver algorithm and implement the concrete model equations using this solver.

Note that an analysis algorithm is appropriate not only for the specific model in question but for at least a whole sub-class of models in the respective

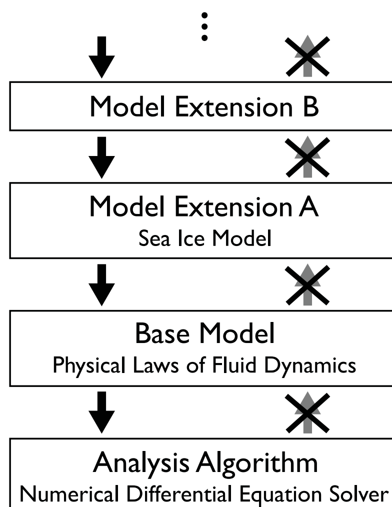

Figure 8.1: Usage relations in the layered architecture of scientific simulation software with examples for an ocean model.

mathematical modeling framework. Therefore, the analysis algorithm can be implemented independently of any concrete model and can be arranged in a way that the model component makes use of the algorithm component but not the other way around.

If additional scientific effects are to be included in the simulation, they can usually be interpreted as extensions to a base model. If, for example, sea ice is supposed to be included in an ocean model, it can be represented as a layer over the entire sea surface which may contain ice of variable thickness [2]. This layer would then influence certain processes which are modeled in the basic fluid dynamics equations.

Such model extensions introduce higher levels of abstraction and can be implemented atop the existing base model, which remains independent of the extension components. In this way, multiple model extensions can be stacked on top of each other, which leads to a layered software architecture as depicted in Figure 8.1.

Alexander and Easterbrook [2] demonstrate that it is not only theoretically possible to employ the multi-layered architecture pattern in the engineering of scientific software but that is actually used by existing simulation software. For this purpose, they analyze the software architecture of eight global climate models that represent both ocean and atmosphere.

Regarding the boundaries between the different components of the climate models, Alexander and Easterbrook point out that they "represent both natural boundaries in the physical world (e.g., the ocean surface), and divisions between communities of expertise (e.g., ocean science vs. atmospheric physics)."

```
#pragma omp parallel for
for(i=1; i<N-1; i++) {
    dt_u[i]   = rho[i];
    dt_rho[i] = (v[i+1] - v[i-1]) / (2*dx);
    dt_v[i]   = (rho[i+1] - rho[i-1]) / (2*dx);
}
```

Listing 8.1: Code snippet from a fictitious C implementation of a finite difference solver for the wave equation.

Therefore, the hierarchically arranged components in simulation software also belong to distinct scientific (sub-)disciplines. This, of course, does not only hold true for climate models but also applies to general simulation software: the analysis algorithm, the base model, and all model extension components are separated from each other along the boundaries of different "communities of expertise."

We will make use of the possibility to partition scientific simulation software along discipline boundaries into hierarchically arranged layers by constructing a *DSL hierarchy* that mirrors this hierarchical structure.

8.3.2 Hierarchies of Domain-Specific Languages

Even though scientific simulation software can typically be engineered using a layered architecture (or actually features such an architecture), "code modularity remains a challenge" [2]. This challenge arises because high performance is required and old programming languages are used (see above). Schnetter et al. [35] demonstrate this with the simple example of a solver for the scalar wave equation in first-order form given as

$$\partial_t u = \rho \tag{8.1}$$

$$\partial_t \rho = \delta^{ij} \partial_i v_j \tag{8.2}$$

$$\partial_t v_i = \partial_i \rho. \tag{8.3}$$

An efficient parallel implementation in C of a finite difference solver for this equation in one dimension would very likely contain a loop like the one in Listing 8.1. It is clearly visible that different concerns are mixed within these few lines of code: the physical model to be simulated (wave equation), the numerical approximation algorithm (finite difference method), and its mapping to hardware resources (memory layout of the vectors, parallelization via OpenMP [12]). A real world application would, of course, be much more complex and would, thus, contain even more intertwined concerns such as memory layout and communication/synchronization for distributed computing nodes. Currently, with low-level programming languages such as C, there

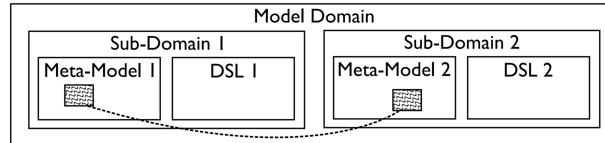

Figure 8.2: Horizontal integration of multiple DSLs. Figure adapted from [38].

is no straightforward way to evade this problem without negatively affecting performance levels.

The aforementioned problems with the modularization of scientific simulation software impede the realization of a layered software architecture that would clearly separate the concerns of different scientific (sub-)disciplines. This makes the software unnecessarily hard to maintain and hinders the cooperation of experts focusing on different scientific aspects of the simulation. Furthermore, it makes it difficult for scientists with only basic programming skills to participate in the development effort at all.

In order to meet these challenges, we propose a software engineering approach called *Sprat*, which is specifically designed for interdisciplinary teams of scientists collaborating on the implementation of scientific (simulation) software. Sprat introduces a DSL for each (sub-)discipline involved in the development project and integrates the languages in a hierarchical fashion based on the layered architectural structure of scientific software outlined above.

8.3.2.1 Foundations of DSL Hierarchies

Typically, DSLs are integrated horizontally as depicted in Figure 8.2 (cf. [38]). In this way, a single domain can be divided into multiple sub-domains that share some common aspects of their respective domain meta-models. Through these shared concepts, the DSLs of the different sub-domains can interact with one another.

For our purpose, however, we need to integrate DSLs from completely different domains (such as numerical mathematics and fish stock modeling). To do so, we extend Stahl and Völter's [38] concept of a *domain-specific platform*. Instead of having a single, pre-implemented platform that already features domain-specific concepts, we introduce multiple, vertically-aligned domain-specific layers that are semantically oriented towards each other as illustrated in Figure 8.3. Each layer is associated with a different DSL which is used to implement a certain part (defined by domain boundaries) of the software system to be constructed. Together, these layers form what we call a *DSL hierarchy*. The layers establish a hierarchy in the sense that at least a portion of the application part associated with each layer forms the (domain-specific) implementation platform for the part on the next higher level. This means that each layer uses abstractions provided by the next lower hierarchy level

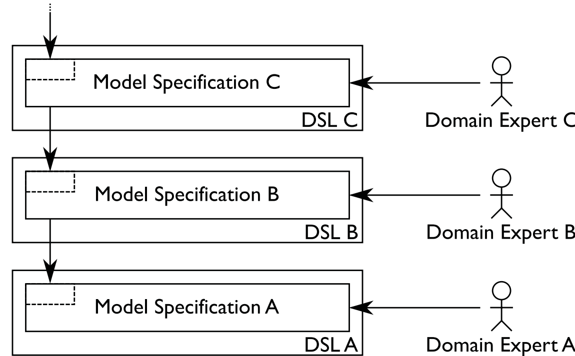

Figure 8.3: Multiple layers acting as domain-specific platforms for each other.

but never uses abstractions from higher levels. For a description of how the levels of the hierarchy can interact and, thus, form domain-specific platforms for each other, see Section 8.3.2.2.

Each level in a DSL hierarchy is associated with a modeler role which uses the DSL of the level to model the application part of this level. Together, the application parts of all hierarchy levels form the whole scientific simulation application to be implemented. Note that we assign a *role* to each level and not a *person*. This implies that a single person can fulfill multiple roles in a DSL hierarchy and one role can be assumed by several persons at once.

By employing an individual DSL for each discipline that is involved in an interdisciplinary scientific software project, we achieve a clear separation of concerns. Additionally, this ensures that all participating scientists (who assume modeler roles) are working only with abstractions that they are already familiar with from their respective domain. Due to the high specificity of a well-designed DSL, the code of an implemented solution that uses this language can be very concise and almost self-documenting. This simplifies writing code that is easy to maintain and to evolve, which allows scientists to implement well-engineered software without extensive software engineering training.

8.3.2.2 An Example Hierarchy

To demonstrate what a DSL hierarchy looks like for an actual scientific software project, we depict in Figure 8.4 the DSL hierarchy for the development of the Sprat Marine Ecosystem Model, which is a PDE-based spatially-explicit model for the long-term prediction of fish stocks. For additional information concerning the different DSLs of the hierarchy and the Sprat Marine Ecosystem Model itself, refer to Section 8.4 below and [22, 23].

The Sprat Marine Ecosystem Model is based on Partial Differential Equations (PDEs) and introduces fish into existing biogeochemical ocean models.

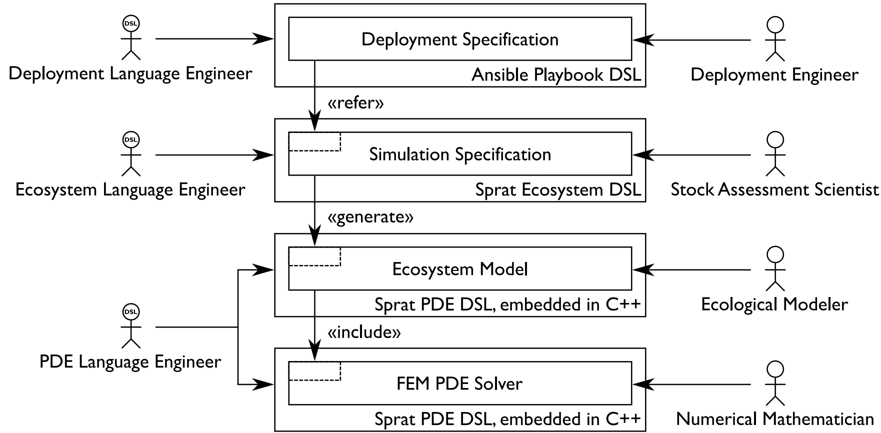

Figure 8.4: DSL Hierarchy for the Sprat Marine Ecosystem Model.

Four different disciplines are involved in the implementation and application of the corresponding simulation software (see the right side of Figure 8.4). At the basis of the hierarchy, we find the role of the Numerical Mathematician who models a special-purpose PDE solver for the model equations. The solver is implemented using the Sprat PDE Solver DSL, which is embedded into C++.

Using the abstractions provided by the bottommost level, the Ecological Modeler implements the concrete equations to be solved for the ecosystem model. Since both the Numerical Mathematician and the Ecological Modeler work with the same abstractions (mathematical equations) to express their application parts, the ecosystem model is implemented with the same DSL as the PDE solver. The interaction between the first and the second layer is of the type *inclusion*: the higher level reuses existing abstractions from the lower level in the same DSL by including them into the model on the higher level.

To apply the simulation to a specific ecosystem (say, the Baltic Sea), it has to be parametrized by a fish stock assessment scientist for that particular ecosystem. For this purpose, the Stock Assessment Scientist creates an ecosystem simulation description using the external Sprat Ecosystem DSL. From such a description, information that is missing for a simulation to be complete is *generated* on the second hierarchy level.

While the first three layers complete the ecosystem simulation as such, it is still undefined how to build and execute the simulation in a (possibly distributed) compute environment. To formally describe this process, the Deployment Engineer models a deployment specification using the external Ansible Playbook DSL [3]. Such a specification interacts with the other levels of the DSL hierarchy by *referring* to names of model artifacts without assuming any knowledge about the internal structure (i.e., the meta-model) of these models.

This level of knowledge is sufficient to, for example, compile application parts.

One could argue that the deployment is a concern orthogonal to the implementation of the simulation and should, hence, not be included in the DSL hierarchy. We decided to incorporate the deployment into the hierarchy nonetheless because it allows us to have a single structure that can be used to abstractly describe to the scientists the whole development process of the simulation up to its execution. This is part of the effort to minimize the accidental complexity of the Sprat Approach (for a more detailed discussion of this aspect, see Section 8.3.4).

The last elements of Figure 8.4 which we have not discussed yet are the language engineer roles. Each DSL has a language engineer role assigned to it that is responsible for the design, implementation, and maintenance of the language. For a description of the individual tasks of a language engineer role and of how to assign this role, see Section 8.3.3.

As with any other approach for bridging the gap between software engineering and computational science, Sprat requires software engineers to assist scientists in the development of scientific software (in our case, language engineers have to design and implement DSLs). This can be problematic because positions for software engineers to provide development support in scientific research institutions have typically not been supported by funding agencies in the past [11]. The neglect of such positions by most funding bodies should be reconsidered, as it has been shown that investing in such positions can have a markedly positive impact [25]. In the meantime, Sprat minimizes the input that is needed from trained software engineers by letting the scientists stay in full control of all development activities of the scientific software itself (using the DSLs designed by professional language engineers; see below).

An overview on the meta-model of our concept of a DSL hierarchy is depicted in Figure 8.5. A detailed description of this meta-model is omitted here for lack of space but the above description of the example hierarchy serves to illustrate the general ideas.

8.3.3 Applying the Sprat Approach

This section describes the engineering process of the Sprat Approach, which builds upon the concept of hierarchies of DSLs introduced above. Sprat acknowledges that computational scientists want to have full control over the implementation of their simulation software. To mediate between the desire for independence on the one hand and the need for assistance on the other hand, the Sprat Approach allows the scientists to continue developing their simulations on their own but with programming languages specifically designed to help them create well-engineered software. Therefore, the Sprat Process, which is shown in Figure 8.6, involves both scientists and DSL engineers [26], with the latter playing a supporting role.

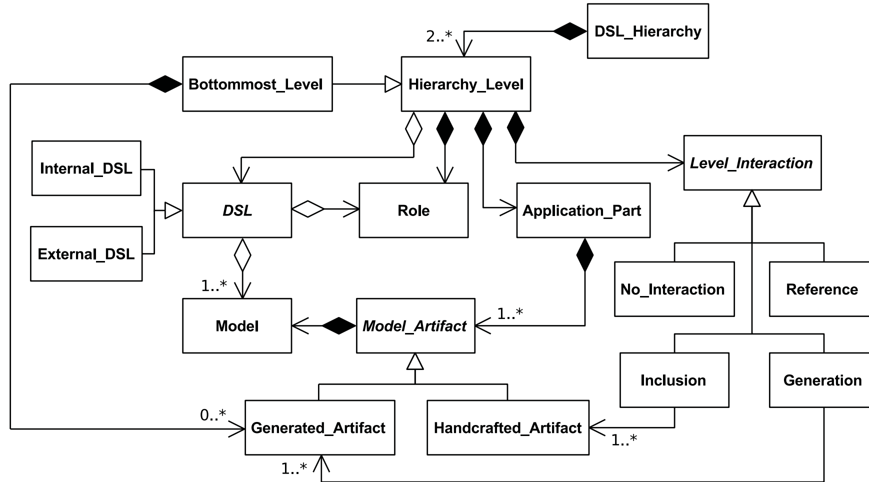

Figure 8.5: Meta-model for the concept of DSL hierarchies using the Unified Modeling Language (UML) notation for class diagrams (attributes of classes not shown, default multiplicity is 1).

8.3.3.1 Separating Concerns

Scientists typically only have a "vague idea" [37] of the simulation software they need for answering their scientific questions. However, such a very general idea is sufficient to construct a DSL hierarchy for the software project. The first step in doing so is for the team of scientists to identify the scientific (sub-)domains that correspond to the classes of scientific effects that need to be modeled. In a second step, these domains are arranged hierarchically as described in Section 8.3.1.

8.3.3.2 Determining Suitable DSLs

Once the levels of the DSL hierarchy and their corresponding application parts have been established, the language engineers must determine whether or not suitable DSLs for the target domains already exist (adopting an existing DSL obviously requires much less effort than creating a new one). For this purpose, Mernik et al. [28] give a collection of patterns that can act as guidelines for deciding whether to develop a new DSL in a given situation. Note, however, that for this activity, the DSL engineers have to take into account a number of factors that are not commonly considered for DSL selection but are of special importance in the context of scientific software development:

1. Many computational scientists are reluctant to adopt "newer" technologies (cf. [16, 33]). Therefore, it has to be ensured that the technologies

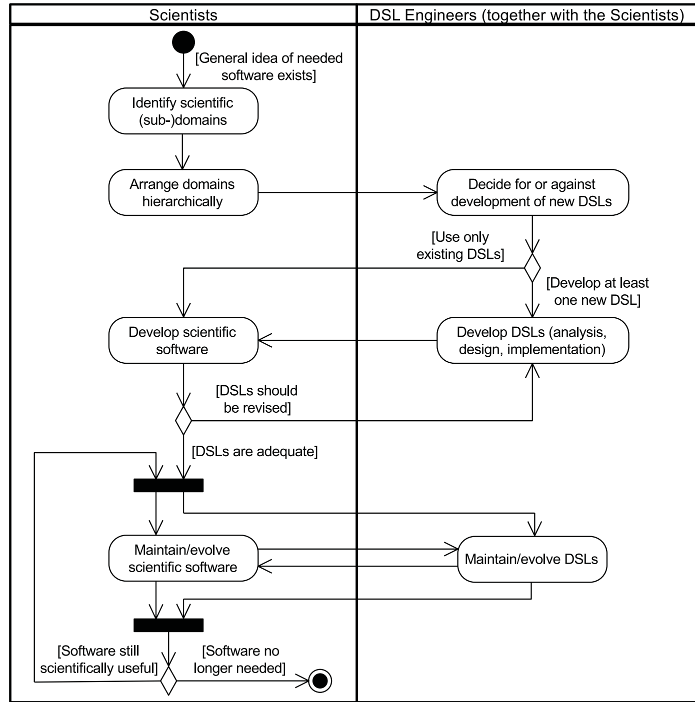

Figure 8.6: Engineering process of the Sprat Approach.

associated with candidate DSLs are accepted among the computational scientists who are supposed to use them.

2. The DSLs have to integrate well with the tools and workflows that the scientists are used to.

3. Candidate DSLs have to be easy to learn for domain specialists (the concrete syntax must appear "natural" to them) and offer good tool support. In this way, the scientists require only minimal training to use the languages.

4. As performance is a very important quality requirement in computational science, it must be made sure that the increased level of abstraction of a candidate DSL does not compromise the runtime performance of programs significantly. Additionally, the DSL should introduce as few dependencies as possible.

5. The language engineers must ensure that candidate DSLs can be integrated with each other vertically in a DSL hierarchy.

Clearly, the language engineers have to cooperate closely with the scientists and obtain feedback from them continuously to make sure that the selected DSLs actually meet the needs of the scientists and that the latter are really willing to use the languages. For this reason, it is important for the language engineers to know about and to respect the characteristics of software development in computational science.

If no suitable DSLs can be identified for some or all levels of the DSL hierarchy, the language engineers have to develop corresponding languages by themselves. In principle, the development of DSLs for computational science is not different from DSL engineering for other domains. Generally, the DSL development process can be divided into a domain analysis, a language design, and an implementation phase for which Mernik et al. [28] identify several patterns. A more detailed approach to DSL engineering that focuses on meta-modeling is given by Strembeck and Zdun [39]. Of course, for DSL development the language engineers have to pay special attention to the same factors that were already discussed above in the context of DSL selection for scientific software development. Again, it cannot be overemphasized that the language engineers have to work in close collaboration with the scientists all the time and that they have to respect (and at least partially embrace) the specific characteristics of scientific software development. For a DSL to be accepted by the target user community, the accidental complexity (both linguistic and technical) introduced along with it must be kept to a minimum.

Concerning the order in which the DSLs should be constructed, we generally propose to develop all languages of the language hierarchy at the same time. Preferably, the development of each DSL takes place in an incremental fashion using agile methods. This approach provides large flexibility because potential incompatibilities between different languages in the DSL hierarchy can be addressed early on. Since DSLs on higher levels of the hierarchy depend on those on lower ones, each development iteration for each language should begin on lower hierarchy levels moving on to higher ones.

8.3.3.3 Development and Maintenance

After DSLs have been assigned to or created for all hierarchy levels, the scientists start to implement the simulation software by assuming the different modeler roles of the DSL hierarchy. The Sprat Process does not impose any restrictions on this activity as this would very likely lead to the rejection of the whole approach (cf. [13]).

If it turns out during the development that some of the DSLs are insufficient for the implementation (e.g., missing elements in the meta-model or an overly technical concrete syntax), the languages have to be adapted by the language engineers. For externally developed DSLs, this very likely means that they have to be replaced by another DSL (possibly one developed "in-house"). This iterative process of the adaptation of the DSLs continues until the simulation software is "finished" in the sense that it can answer the sci-

entific questions it was designed for (or the ones that emerged along the way during the implementation).

After reaching a state of relative maturity and stability, the simulation software enters its maintenance phase. In this phase, the number of changes applied to the software per unit of time is typically much lower than during the initial implementation phase. Note, however, that especially in computational science, the boundaries between the development and maintenance phase are rarely clear-cut.

During the maintenance phase, the simulation software is evolved in order to enable answering new scientific questions. Typically, the DSLs of the hierarchy should be able to support the changes to be introduced to the simulation. However, if previously ignored aspects of a domain have to be included in the simulation, also the DSLs have to be evolved in parallel to the scientific software.

New scientific questions could further make it necessary to add new levels to the DSL hierarchy because it may be required to model effects from totally different domains. In this case (which is not depicted in Figure 8.6 for reasons of clarity), one would have to start with the decision for or against the development of a new DSL for this level. The rest of the process for this specific hierarchy level would be the same as for the other levels.

After each maintenance iteration, when the new scientific questions could—or could *not*—be answered, the question arises whether it is still scientifically useful to maintain the simulation software. Depending on the answer to this question, either a new maintenance iteration is started or the software is not developed any further and the Sprat Process comes to its end.

8.3.4 Preventing Accidental Complexity

Segal [36] reminds us that software engineers "should not try to impose the full machinery of traditional software engineering on scientific software development." Any tool or development approach which assumes that scientific programmers will invest time and effort into mastering it is deemed to fail because "scientists tend to want results immediately" [31]. Therefore, Prabhu et al. [31] conclude that while educating scientists in software engineering methods is worthwhile, "a more promising approach is to develop solutions that are customized to the requirements of scientists" and "require little training." Such solutions have to adopt the frame of reference of the scientists and must necessarily make compromises with regard to their generality and formality [24]. If a tool confronts scientists with too many formal software engineering complexities—which might seem natural for a software engineer but are "accidental" from a scientist's perspective—, the tool will inevitably face rejection [40].

The Sprat Approach achieves a compromise between formality and pragmatism by making two central concessions. First, we do not impose any restrictions on the concrete development activities of the scientific programmers, as

discussed in Section 8.3.3.3. Second, we refrain from too much formality in the artifacts that are necessary for carrying out a scientific software development project with the Sprat Approach.

The only artifact that the scientists produce together with the language engineers to communicate the development process among themselves is a diagram of the DSL hierarchy. Therefore, *all* development aspects have to be represented in this diagram. This includes even concerns that could be modeled as orthogonal to the actual development of the software, such as the deployment process (cf. Figure 8.4). Thus, the hierarchy diagram represents a combination of different concerns and even mixes structural and procedural elements (e.g., x must be present before y can be deployed). This approach minimizes the complexity that the scientific programmers have to deal with but still enables meaningful reasoning about the software, its development process, and the different responsibilities of the personnel involved.

8.4 Case Study: Applying Sprat to the Engineering of a Coupled Marine Ecosystem Model

We evaluated the Sprat Approach by conducting a case study in which we applied Sprat to engineer a coupled marine ecosystem model that has been developed in collaboration with GEOMAR Helmholtz Centre for Ocean Research Kiel and Dalhousie University. In this section, we give a brief overview on the ecosystem model itself as well as on the DSLs used to implement it before discussing the results from our case study in Section 8.5. For a more in-depth description of the Sprat Marine Ecosystem Model and the DSLs utilized in its implementation, see [22, 23].

8.4.1 The Sprat Marine Ecosystem Model

The Sprat Marine Ecosystem Model is a PDE-based ecosystem model for long-term fish stock prediction that is coupled with existing biogeochemical ocean models. This online coupling allows to study the interactions of the different trophic levels in marine food webs.

Based on so-called population balance equations [32], the model's central system of PDE for $n \in \mathbb{N}$ fish species is given by

$$\frac{\partial}{\partial t}u^{[\kappa]} + \sum_{i=1}^{d} \frac{\partial}{\partial x_i} q_i^{[\kappa]} u^{[\kappa]} + \frac{\partial}{\partial r} g^{[\kappa]} u^{[\kappa]} = H^{[\kappa]} \tag{8.4}$$

with $\kappa = 1, \ldots, n$. Here, $u : \mathbb{R}_{\geq 0} \times \Omega_{\mathrm{x}} \times \Omega_{\mathrm{r}} \mapsto \mathbb{R}^n$ represents the time-dependent mass distribution of fish in space and size/weight dimension. Thus, for every point in time $t \geq 0$, $u^{[\kappa]}$ assigns each point in space $x \in \Omega_{\mathrm{x}} \subset \mathbb{R}^d$ and each

possible size of a fish $r \in \Omega_r \subset \mathbb{R}_{>0}$ the average mass $u^{[\kappa]}(t, x, r)$ of individuals from species κ present with these coordinates. The $q_i^{[\kappa]}$ are spatial movement velocities and $g^{[\kappa]}$ is a growth velocity; all of them depend non-linearly on u. $H^{[\kappa]}$ is the forcing term that abstracts the sources and sinks of individuals (e.g., birth or fishing).

The DSL hierarchy for implementing this model with the Sprat Approach, which has already been presented in Section 8.3.2.2, employs three DSLs:

1. The Sprat PDE DSL for implementing the numerical solver and the equations of the Sprat Model.
2. The Sprat Ecosystem DSL for specifying ecosystem simulation experiments.
3. The Ansible Playbook DSL for describing the deployment of the Sprat Model in distributed High Performance Computing (HPC) environments.

The first two of these languages were developed by us while the third one was reused.

8.4.2 The Sprat PDE DSL

The Sprat PDE DSL is embedded into C++ via template meta-programming techniques [1] and focuses on the implementation of special-purpose Finite Element Method (FEM) PDE solvers [8]. Since it targets developers of such algorithms rather than FEM practitioners, the language does not feature the most abstract concepts of FEM (say, variational forms) but concentrates on entities that allow to conveniently write mesh-based PDE solvers. From a technical perspective, the language is comprised of a set of header files written in C++11 that can be used by the application programmer with just a single include statement. These headers expose a set of classes and functions to implement the following three key features which are illustrated in Listing 8.2:

1. Lazily evaluated matrix-vector arithmetic with a natural and declarative syntax (see, e.g., Line 12)
2. Iterations over sets (see Lines 6–8)
3. Single Program, Multiple Data (SPMD) abstractions for parallelization (see Lines 14–15)

8.4.3 The Sprat Ecosystem DSL

In order to apply the fish stock model implemented with the Sprat PDE DSL, it has to be parameterized for a specific marine ecosystem by a stock

```
1  DistributedVector  u, q;
2  ElementVectorArray F_L;
3  ElementMatrixArray C;
4  ElementMatrix      D;
5
6  foreach_omp(tau, Elements(mesh), private(D), {
7      foreach(i, ElementDoF(tau), {
8          foreach(j, ElementDoF(tau), {
9          D(i, j) = max(i.globalIndex(), j.globalIndex());
10         })
11     })
12     F_L[tau] = C[tau]*q + D*u;
13 })
14 u *= u.dotProduct(q);
15 u.exchangeData();
```

Listing 8.2: Sprat PDE DSL code snippet.

assessment scientist. For this purpose, we used Xtext [15, 14] to implement the external Sprat Ecosystem DSL that allows to specify the properties of the simulation run as well as the ecosystem and its fish species in an abstract and declarative way.

Figure 8.7 shows an example of a simulation description using the Sprat Ecosystem DSL in our Integrated Development Environment (IDE) for the language with custom syntax coloring. A simulation description consists of several top-level entities (Ecosystem, Output, Input, Species) that possess attributes which describe the entity. Most of these attributes have a constant numerical value given by an expression with a unit. If a unit is missing, the editor issues a warning and offers a quick fix that adds a unit of the correct quantity category to the expression. Unit conversions (e.g., from degree Fahrenheit to degree Celsius) are automatically carried out by the DSL. A keyword of the language is `record`, which can be used in the Output entity to let the user describe which data should be collected during a simulation run. This allows to aggregate the information already while the simulation is running and thus makes it unnecessary to store the potentially huge amount of all data generated by the simulation.

8.4.4 The Ansible Playbook DSL

Once the Sprat Model is fully parametrized by a Sprat Ecosystem DSL specification, a last piece of information is missing for a complete executable simulation: a description is needed of how to deploy it in a distributed HPC environment. For this purpose, we reuse the Ansible Playbook DSL [3], which describes configuration states that certain computer systems are supposed to be in. The user neither has to specify the initial state of the system nor

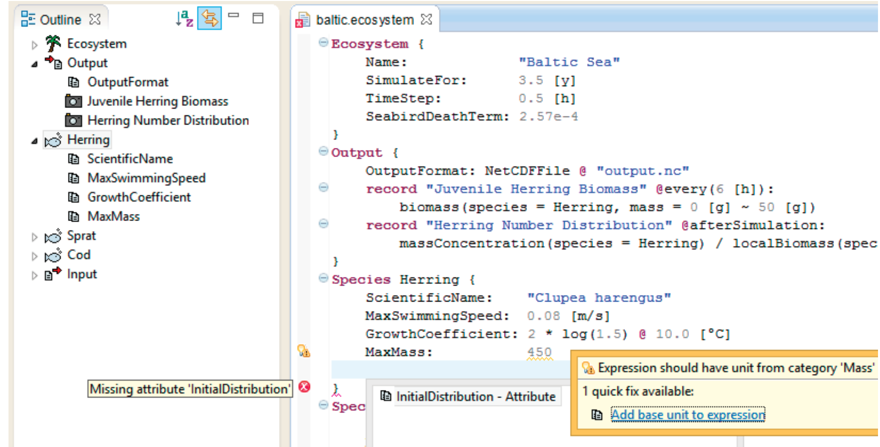

Figure 8.7: IDE for the Sprat Ecosystem DSL.

the transformations that have to be applied to achieve the desired state. An example of the syntax of the DSL is given in Listing 8.3.

8.5 Case Study Evaluation

We employed a mixed-methods research design to evaluate the application of the Sprat Approach for the development of the Sprat Model. Specifically, we used online surveys, controlled experiments, performance benchmarking, and expert interviews to assess the quality of the DSLs and their impact on the development process. To save space, only results from the expert interviews that we conducted in order to evaluate the Sprat PDE DSL are discussed here. These results allow us to formulate some general lessons learned for developing DSLs for computational scientists.

8.5.1 Data Collection

We conducted semi-structured interviews [21] with eight experts which each lasted between one and two hours. The interviews were mostly carried out in a one-on-one fashion (with the exception of one interview in which we talked to two experts) in the expert's workplace.

Half of the sample were domain experts for PDE solvers and the other half were professional DSL developers. The domain experts were recruited by contacting professors of relevant research groups at Kiel University and

```
---
- hosts: localhost
connection: local
vars_files:
- ./openstack_config.yml

tasks:
- name: Make sure the cloud instances are running
nova_compute:
state:      present
hostname:   sprat{{ item }}
image_id:   "{{ os_image_id }}"
with_sequence: start=1 end={{ nInstances }}
```

Listing 8.3: Excerpt from the Ansible Playbook for deploying the Sprat Simulation on an OpenStack cloud.

GEOMAR Helmholtz Centre for Ocean Research Kiel. Candidates for the professional DSL developer group were selected from local industry. The actual involvement of these persons in DSL development was verified prior to conducting the interviews.

The group of domain experts consisted of one postdoc and three professors from research areas such as numerical analysis, optimization, and ocean modeling (all from different research groups). The professional DSL developers all had multiple years of experience with DSL design (one of them even authored a book on MDSE and DSLs) and were IT architects or consultants.

Two interview guides (one for the domain experts and one for the DSL developers) were developed based on four analysis dimensions deduced from our research question: *how do experts rate the functional and technical quality of the Sprat PDE DSL?* These analysis dimensions are:

1. Software development in computational science (only for the domain experts)
2. Learning material for DSLs
3. Meta-model and syntax of the Sprat PDE DSL
4. Technical implementation of the Sprat PDE DSL

Prior to the interviews, all interviewees were supplied with the source code of, code examples for, and a short written introduction to the Sprat PDE DSL in order to familiarize themselves with the language. Since we expected that not all of the interviewees would find the time to have a closer look at the language beforehand, we started each interview by briefly discussing the fundamental concepts of the language and by going through example algorithms implemented with the DSL. To make sure that the experts themselves had

actually worked with the language, each interviewee was asked to solve tasks related to the supplied code examples during the interview (e.g., changing the mesh of a solver or altering the parallelization scheme of an algorithm).

8.5.2 Analysis Procedure

All interviews were taped and fully transcribed prior to the analysis. The transcripts were analyzed using the method of *qualitative content analysis* (specifically, *summarizing content analysis*) [27], for which we chose whole answers as our unit of analysis.

8.5.3 Results from the Expert Interviews

In the following, we describe selected aspects of the interview results. We present only results that address questions of DSL design for computational science in general and omit results that focus only on the quality of the specific DSL in question. This approach allows us to present some broader conclusions and lessons learned in Section 8.6.

8.5.3.1 Learning Material for DSLs

In this section, we discuss what kind of learning material the domain experts wish for in order to familiarize themselves with DSLs for computational science in general and with the Sprat PDE DSL in particular and how their requests differ from the ideas of the professional DSL developers.

All the scientists view commented example programs as the key element for the introduction to a new DSL for three reasons:

1. Example programs allow to judge quickly whether the DSL is suitable for the intended application and whether the scientists can imagine to work with it ("Is the code compact? Does it seem intuitive? Do I understand it?").

2. Examples make it easy to learn how a typical DSL program is structured.

3. Examples can be used as a basis for own programs without investing much time in reading other documentation artifacts.

Complementary to a set of commented examples, the interviewees would like to have a summary of the key concepts of the language which answers questions such as: What is the exact scope of the DSL? What is the performance of matrix-vector expressions? How is data managed with the language? Additionally, they would like to have a specification of the data structures and interfaces of the DSL in order to understand how (possibly already existing) GPL code can be combined with the constructs of the language.

The DSL developers also suggest a combination of a summary and a complete language reference as learning material. However, they do not mention

the importance of complete code examples (they rather seem to think of example snippets embedded into written text) and they generally put much more emphasis on the reference document than the scientists do: three out of four interviewed DSL developers name the reference first and talk about introductory material only when asked about it. For them, the basis of the reference document should be a formalized meta-model or the abstract syntax of the DSL, which is supposed to quickly give a top-down overview on the language. One of the interviewees mentions the reference of the Swift programming language [4] as exemplary in this respect. This reference is structured around grammar rules that are grouped according to which aspect of the language they belong to (expressions, types, etc.).

From the interviews, it can be seen that the domain scientists seem to favor a more practical and pragmatic approach to learning a DSL for computational science than DSL designers might think. The scientists emphasize the importance of complete documented code examples and they are interested in a reference only as a second step when it comes to more technical aspects of the implementation. In consequence, the utility of a formalized meta-model and lengthy grammar rule descriptions seems questionable for such an audience. When developing DSLs for computational scientists, DSL designers should reflect on their generally more formal and systematic approach to introducing others to such a language.

8.5.3.2 Concrete Syntax: Prescribed vs. Flexible Program Structure

One of the experts in DSL development suggests to enforce a common block structure for all Sprat PDE DSL programs. His motivation is to make sure that "as little nonsense as possible happens." When two of the domain experts are confronted with this idea in two subsequent interviews, they both express their fear that a prescribed code structure would take away too much control over their program and would make integration with existing code harder.

While software engineers working in the IT industry generally seem to focus their attention on consistency among solutions to facilitate reusability and maintainability, computational scientists favor loose structures that allow them to experiment and quickly obtain results. In order to be accepted by the scientists, a DSL for computational science (and especially for the HPC community) has to be pragmatic about the rigidity of prescribed structures and the level of abstraction it introduces. If one does not pay concessions to the need of computational scientists to freely experiment with a DSL, the language simply will not be adopted.

8.5.3.3 Internal vs. External Implementation

All except one of the domain experts favor an internal DSL over an external language for the level of abstraction that the Sprat PDE DSL aims for. They

name several reasons for this:

1. An external DSL with a completely new syntax could be in conflict with concepts of programming languages such as Fortran and C that the experts have internalized. This could lead to confusion and an increased number of errors.

2. One expert already has negative experiences with external DSLs for implementing numerical algorithms. He says that such languages are often very good for the narrow domain they are designed for but commonly lack support for "everything else" in the large domain of computational science. In contrast to this, with an internal DSL, the user can seamlessly integrate DSL code with GPL code.

3. Another interviewee states that he does not believe that the increased flexibility of an external DSL offsets the added technical complexity of additional compiler runs and the need for other external tooling.

In spite of these reservations, none of the interviewees excludes the use of an external DSL with a code generator categorically, as long as this code generator is portable and available under an open source license.

The single domain expert who would actually prefer the Sprat PDE DSL to be an external language works with ocean models. He reports that in this scientific field, researchers are sometimes confronted with the problem of not being able to reproduce older simulation results once hardware platforms and compiler vendors/versions change. Therefore, he is interested in archiving source code that is as low-level as possible (e.g., already preprocessed Fortran code). With an internal language which uses template meta-programming techniques, such as the Sprat PDE DSL, this is not possible. Template meta-expressions are processed during compilation without yielding any intermediate low-level C++ code that could be archived. This lack of intermediate code also makes debugging of matrix-vector expressions hard because one cannot see what is actually executed during the evaluation of such an expression.

Since nobody excluded the use of an external DSL completely, a compromise would be possible: one could implement the Sprat PDE DSL as an external *language extension* to C or C++ using a framework such as the Meta Programming System (MPS) [10]. In this way, the requirements of both proponents, of those of an internal and of those of an external solution, could be met. All the features of the C/C++ GPL would be present while readable intermediate C/C++ code could be generated. It would even be possible to incorporate *interactive model-based compilation* techniques into the tooling that allow to trace in detail the transformations applied to DSL models during code generation [30]. However, further research has to be conducted in order to evaluate whether such an approach would actually be accepted in the community.

8.6 Conclusions and Lessons Learned

By assigning a DSL to each scientific (sub-)discipline that is involved in an interdisciplinary effort to develop complex simulation software, Sprat achieves a clear separation of concerns and enables scientists to produce maintainable, performant, and portable implementations by themselves without the need for extensive software engineering training. To evaluate the Sprat Approach, we applied it to the engineering of the Sprat Marine Ecosystem Model, which allows us to establish important lessons learned for the design of DSLs for computational science. These lessons learned, which stem mainly from the expert interviews conducted in the context of our case study, concern four areas of DSL design:

1. *Abstraction level of the meta-model*: the more related the application domain of a DSL is to computation, the more it needs to be possible for the users to influence how computations are executed (i.e., the closer to the underlying hardware platform the language must be). This especially allows full control over the runtime performance of solutions. In our experience, such languages are best implemented as internal DSLs embedded in relatively low-level programming languages, such as C++, which enables to reuse much of the existing language facilities of the host language.

2. *Concrete syntax*: the scientists participating in our study favor DSLs that do not prescribe too much structure of models because they want to have full control over their code and want to be able to experiment with it quickly and freely.

3. *DSL tooling*: in application domains not related to computing (such as biology) external DSLs with their own tooling are more likely to be accepted. The tooling should make sure that only scientifically reasonable DSL models are permitted and, otherwise, confront the user with error messages on the abstraction level of the domain (which would not be possible after generating code into programming languages with a lower abstraction level).

4. *Documentation*: professional DSL designers often focus on language references based on a formal meta-model or on the abstract syntax while scientists favor code examples to get a fast overview on the capabilities of a DSL and to become productive quickly.

In the future, we plan to extend the Sprat Approach by including model-based software performance engineering and testing techniques [34, 5]. Since with Sprat, all implementation artifacts already are high-level models, these techniques would allow to increase the runtime performance and credibility of

simulation results without considerable additional effort for the computational scientists.

Bibliography

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

[2] K. Alexander and S.M. Easterbrook. The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations. *Geoscientific Model Development Discussions*, 8(1):351–379, 2015.

[3] Ansible Incorporated. Ansible documentation. `http://docs.ansible.com`, 2015.

[4] Apple Incorporated. The Swift programming language – language reference. `https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html`, 2015.

[5] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *Software Engineering*, 30(5):295–310, 2004.

[6] Victor R. Basili, Daniela Cruzes, Jeffrey C. Carver, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Marvin V. Zelkowitz, and Forrest Shull. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008.

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Number 1 in Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012.

[8] Susanne Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer, 3 edition, 2008.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. Pattern-oriented software architecture, volume 1: A system of patterns, 1996.

[10] Fabien Campagne. *The MPS Language Workbench: Volume I*. Fabien Campagne, 2014.

[11] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 550–559. IEEE, 2007.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering*, 5(1):46–55, 1998.

[13] S.M. Easterbrook and T.C. Johns. Engineering the software for understanding climate change. *Computing in science & engineering*, 11(6):65–74, 2009.

[14] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, pages 112–121. ACM, 2012.

[15] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object-oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[16] Stuart Faulk, Eugene Loh, Michael L. Van De Vanter, Susan Squires, and Lawrence G. Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11:30–39, 2009.

[17] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[18] V. Grimm and S.F. Railsback. *Individual-based Modeling and Ecology*. Princeton University Press, 2005.

[19] J.E. Hannay, H.P. Langtangen, C. MacLeod, D. Pfahl, J. Singer, and G. Wilson. How do scientists develop and use scientific software? In *Software Engineering for Computational Science and Engineering, 2009. SECSE'09. ICSE Workshop on*, pages 1–8. IEEE, 2009.

[20] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207–219, 2015.

[21] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE International Software Metrics Symposium (METRICS 2005)*, pages 1–10. IEEE, 2005.

[22] Arne N. Johanson and Wilhelm Hasselbring. Hierarchical combination of internal and external domain-specific languages for scientific computing. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW'14, pages 17:1–17:8. ACM, 2014.

[23] Arne N. Johanson and Wilhelm Hasselbring. Sprat: Hierarchies of domain-specific languages for marine ecosystem simulation engineering. In *Proceedings TMS SpringSim'14*, pages 187–192. SCS, 2014.

[24] Diane Kelly. A software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6):120–119, 2007.

[25] Sarah Killcoyne and John Boyle. Managing chaos: Lessons learned developing software in the life sciences. *Computing in Science & Engineering*, 11(6):20–29, 2009.

[26] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 2008.

[27] Philipp Mayring. *Qualitative Inhaltsanalyse: Grundlagen und Techniken*. Beltz, 12 edition, 2015.

[28] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[29] Paul Messina. Gaining the broad expertise needed for high-end computational science and engineering research. *Computing in Science & Engineering*, 17(2):89–90, 2015.

[30] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. Compiling SCCharts – a case-study on interactive model-based compilation. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 461–480. Springer, 2014.

[31] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A survey of the practice of computational science. In *State of the Practice Reports*, SC'11, pages 19:1–19:12. ACM, 2011.

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

[33] Rebecca Sanders and Diane F. Kelly. Dealing with risk in scientific software development. *Software, IEEE*, 25(4):21–28, 2008.

[34] Ina Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, 2012.

[35] Erik Schnetter, Marek Blazewicz, Steven R. Brandt, David M. Koppelman, and Frank Löffler. Chemora: A PDE-solving framework for modern high-performance computing architectures. *Computing in Science & Engineering*, 17(2):53–64, 2015.

[36] Judith Segal. Models of scientific software development. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering, SECSE'08*, pages 1–7, 2008.

[37] Judith Segal and Chris Morris. Developing scientific software. *Software, IEEE*, 25(4):18–20, 2008.

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

[39] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[40] Gregory V. Wilson. Where's the real bottleneck in scientific computing? *American Scientist*, 94(1):5–6, 2006.

[41] Gregory V. Wilson. Software carpentry: lessons learned. *F1000Research*, 3:1–11, 2014.