View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OceanRep

Hierarchical Combination of Internal and External Domain-Specific Languages for Scientific Computing

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

ABSTRACT

To adapt established methods of software engineering for scientific computing, we propose a software development approach for interdisciplinary teams of scientists called Sprat. The approach is organized around a hierarchical architecture that combines internal and external domain-specific languages (DSLs). For its evaluation, Sprat is employed in the implementation of a marine ecosystem model. We highlight what is to be observed while integrating the DSLs into the hierarchy in order to enable a successful cooperation of scientists in interdisciplinary teams as well as to achieve a maintainable code base.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

1. INTRODUCTION

With the ever-increasing computing power, *in silico* experiments are becoming more complex. Not only can computational scientists and modelers simulate phenomenona with a much higher resolution but they can also include more and more details into the models themselves to improve their predictive capabilities [7]. As a consequence, the complexity and lifespan of scientific software is growing as well as the necessity for its output to be reproducible and verifiable. Additionally, there is an increasing need for collaboration between scientists from different disciplines to create such complex software systems.

To meet these challenges efficiently, the developers of scientific software must produce a maintainable and testable code base. But because these developers are usually the scientists themselves and because of the "wide chasm" [8] between the disciplines of scientific computing and software engineering, most of these scientists have not received any training in software engineering that would have taught them established tools and best practices to achieve the aforementioned 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. As a result of this, and because of the different role that software plays in the scientific community [4], software engineering tools and best practices cannot simply be transferred to computational science but have to be adapted to this domain.

In response to these challenges, we propose a software engineering approach specifically tailored to the needs of interdisciplinary teams of scientists [9]. The approach, which we named Sprat, is organized around an architectural design based on hierarchies of domain-specific languages (DSLs). This design allows scientists from different disciplines to collaborate on the implementation of scientific software, utilizing only the abstractions they are familiar with from their respective domain. Thanks to the high-level domain concepts, the resulting source code is very concise and thus easy to maintain and test as well as almost self-documenting. As an evaluation scenario for the Sprat approach, we chose the implementation of a spatially explicit partial differential equation (PDE)-based ecosystem model for fish stock prediction.

While in [9] we give an overview of the Sprat approach, this present paper will describe in detail the architectural design of the DSL hierarchy and the interactions of the languages in this hierarchy. For this purpose, Sect. 2 recapitulates the foundations of the Sprat approach – namely, the notion of DSL hierarchies – and describes its practical application in general. After these more conceptual considerations, we focus on the application of our approach to our evaluation scenario (Sect. 3). Section 4 concentrates on the three DSLs employed in the implementation of the ecosystem model. We highlight what is to be observed while integrating the DSLs into the hierarchy in order to enable a successful cooperation of scientists in interdisciplinary teams as well as to achieve a maintainable code base. In particular, we will see that it is beneficial to use internal DSLs on the lower layers of the hierarchy and to adopt external DSLs on higher layers for high-level abstractions. We conclude the paper with a discussion of related work (Sect. 5) and a summary of the primary insights as well as further research directions (Sect. 6).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ECSAW August 25 - 29 2014, Vienna, Austria
Copyright 2014 ACM 978-1-4503-2778-7/14/08
http://dx.doi.org/10.1145/2642803.2642820 ...$15.00.

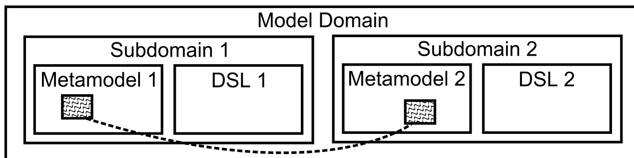

Figure 1: Typically, domains (and thus systems) are split horizontally. Figure adapted from [18].

2. THE SPRAT APPROACH: HIERARCHIES OF DSLS

The Sprat approach is based on techniques of model-driven software development (MDSD). It adopts the key concept of MDSD to generate automatically as many artifacts of a software product as possible from an abstract representation – a model – of the solution to be implemented. The model can be represented either graphically or textually by a domain-specific language (DSL). A DSL is tailored to a specific application domain and thus trades the generality of a general-purpose language (GPL) like C or Java for an increased conciseness in that domain. DSLs are referred to as *internal* or *embedded* if they are implemented as libraries of a general-purpose language but appear to be independent languages. In contrast to this, *external* DSLs are completely independent languages with their own syntax and semantics [18].

Concerning the concept of MDSD, we follow Stahl and Völter [18] who promote a pragmatic approach to model-driven methods by focusing less on model-to-model transformations but more on code-generation from a model. As a consequence, they add to the model – the first cornerstone of MDSD – the concept of a "semantically rich" domain-specific platform as a second key element. Because such a platform already incorporates reusable domain-specific components, model-to-code transformations are simplified.

Building on the notion of transformations between layers that are semantically oriented towards each other, we introduce the concept of *hierarchies of DSLs* by combining several of these layers 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. 2 in Sect. 3, which illustrates this relation for our evaluation example). This vertical hierarchy of DSLs is in contrast to the way multiple DSLs are usually organized within a project: typically, they form a horizontal structure that divides the targeted domain in sub-domains [18]. In this case, the DSLs have to share some common aspect of the domain metamodel which allows them to interact with each other (cf. Fig. 1).

The Sprat approach is organized around a DSL hierarchy in the sense described above. Such a vertical hierarchical structuring of DSLs lends itself to a development method for scientific software systems because it resembles the dependency of the sub-systems contributed by specific disciplines. Similar to our evaluation example (cf. Sect. 3), it is often the case that the sub-systems implemented by experts from a specific domain form a strict usage or realization hierarchy. This means, that sub-systems that are on a higher level make use of or rather are realized by the underlying sub-systems but never the other way around (e.g., a biological model uses/is realized by a certain numerical solver but not vice versa).

By employing multiple DSLs organized in this way, different systems that build upon each other can all be implemented by different developer roles from various domains. In the context of scientific software development, this implies that scientists from different areas are enabled to collaborate on software projects while only working with abstractions they are familiar with from their respective domains. Due to the high specificity 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 [11].

2.1 Applying the Sprat Approach

To employ the Sprat approach in an interdisciplinary effort for implementing a scientific software solution, the first step is to identify the different coarse sub-systems (and thus DSLs) that will be needed for the project at hand. Once the target domains for which languages are to be developed have been identified, initial versions of the DSLs have to be constructed (if there are no existing DSLs that suit the domain). In this step, a software language engineer – who represents an additional role in our approach – will work in close cooperation with the respective domain experts. Among other things, the language engineer is responsible for selecting an appropriate technology for the DSL to be implemented in and has to ensure a good integration with the other DSLs of the hierarchy. While doing so, it is vital *not* to treat the selection of the implementation technology for the language as merely a technical issue. For such a language to be accepted by the targeted user community, no "accidental complexity" [21] should be introduced along with it. This means, that the technology of the DSL has to integrate well with the workflows and technologies that are established in the targeted domain.

Concerning the order in which the DSLs are created, we generally propose to incrementally develop all languages at the same. This is because the generator of each DSL depends on the DSL of its subordinate layer. As a result of this dependency, it will usually prove useful to begin a development iteration on lower moving on to higher layers.

While designing the concrete syntax of the DSL, the software engineer must pay attention to making the language very easy to learn for the targeted user group. If the syntax is too complex and does not appear "natural" to the domain experts, the DSL might be perceived as just another "accidental complexity."

For the same reason, full tool support for the DSLs has to be provided. And finally, it should be ensured that the increased level of abstraction does not compromise the runtime performance of programs significantly (a requirement that is especially important for the high-performance computing (HPC) community) and that as few dependencies on libraries, etc. as possible are introduced.

Apart from the considerations discussed above, the design and implementation of the languages follows the well-established DSL life cycle described in [12].

3. EVALUATION SCENARIO: SPRAT MARINE ECOSYSTEM MODEL

In order to evaluate the Sprat approach and to see how the

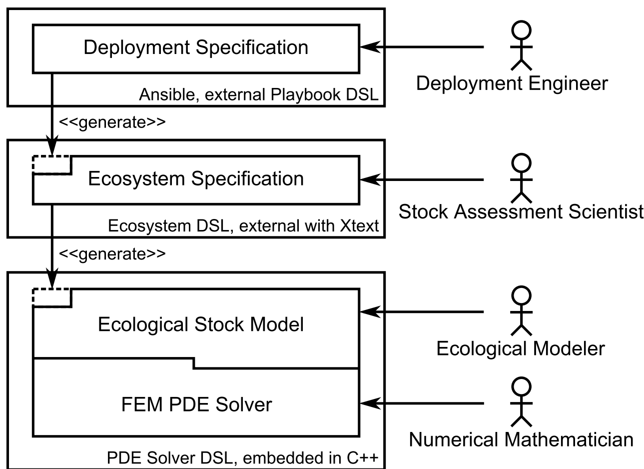

Figure 2: Artifacts and roles associated with the different DSLs of the Sprat simulation.

benefits of hierarchically organized DSLs for scientific application development can be obtained in practice, we apply the method to the implementation of the *Sprat model.*

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

Based on an advection-diffusion or population balance equation, the model's central system of PDEs for $n \in \mathbb{N}$ fish species is given by

$$\frac{\partial}{\partial t} u^{[\kappa]} + \sum_{i=1}^{d} \frac{\partial}{\partial x_i} q_i^{[\kappa]} u^{[\kappa]} + \frac{\partial}{\partial r} g^{[\kappa]} u^{[\kappa]} - \varepsilon \Delta u^{[\kappa]} = H^{[\kappa]} \quad (1)$$

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 average number 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 $r \in \Omega_{\mathrm{r}} \subset \mathbb{R}_{>0}$ the average number $u^{[\kappa]}(t, x, r)$ of individuals from species κ present with these coordinates. $q_i^{[\kappa]}$ and $g^{[\kappa]}$ are the spatial movement velocities and the growth velocities respectively that depend non-linearly on u. $H^{[\kappa]}$ is the forcing term that abstracts the sources and sinks of individuals (e.g., birth or fishing). If appropriate, a small amount of diffusion ($\varepsilon \Delta u^{[\kappa]}$, where ε is close to zero) may be added to the model to incorporate random movement of the fish.

3.1 The DSL Hierarchy for the Sprat Model Implementation

As described in Sect. 2.1, the first step to apply the Sprat approach is to identify the disciplines involved in the project and their respective contributions of sub-systems. Based on these contributions, we define the developer roles and DSLs in the hierarchy as shown in Fig. 2.

At the basis of the hierarchy, we find the role of the numerical mathematician who implements a solver for the model equations (1). Building upon this solver, an ecological modeler constructs a concrete version of the model (which includes specifying q, g, and H). Both roles can share a common programming language – the Sprat PDE solver DSL – as both express their results in "mathematical formulas." In order to implement the PDE solver DSL, we chose to embed the language into C++. The reason for embedding the DSL into a general-purpose language is that the numerical mathematician must have such a language at hand because of the constant need to implement problem-specific datastructures. In this way, we also obtain full tool support for the general-purpose language and do not have an additional generation step. C++ as the host language was mainly selected 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.

The concrete version of the model implemented with the Sprat PDE solver DSL is used by fish stock assessment scientists to run simulations parameterized for a specific ecosystem. Thus, – among other things – they have to specify details of the fish species found in that system, the topography of the ocean region, as well as which parameters to aggregate and record during the simulation. To design a concise DSL that fulfills these requirements, full control over its concrete syntax in necessary. That is why the Sprat ecosystem DSL is implemented as an external DSL using Xtext [5]. Xtext is based on the Eclipse Modelling Framework [19] and lets the user specify a grammar in a notation similar 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 autocomplete feature are generated. Thereby, complete tool support for the DSL is achieved easily.

Once a simulation is fully parameterized, it is still lacking a last element in order to be complete: a deployment engineer needs to be able to specify as abstractly as possible how the simulation is to be executed in a (distributed) computing environment. For this purpose, we employ an existing DSL that fits this use case well, namely we reuse Ansible[1] and its external "Playbook Language."

Because each developer role uses only one of these hierarchically ordered DSLs, a clear separation of concerns and thus a clear modularization of the software is achieved. In our evaluation example, however, there is one noteworthy exception from this as the artifacts produced by the numerical mathematician and the ecological modeler overlap. 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 explicitly.

4. THE DSLS EMPLOYED IN THE HIERARCHY

In the following, the implementation of the DSLs that have been introduced in the previous section is discussed. We highlight important design considerations for the DSLs that are crucial to achieve the benefits of the Sprat approach and its hierarchical DSL architecture that have been mentioned in the introduction.

[1] `http://www.ansible.com`

4.1 The Sprat PDE Solver DSL

We already discussed the design considerations that led us to embed the Sprat PDE solver DSL into C++. In this section, we describe the features of the language, the technical aspects of its implementation, and how both contribute to its conciseness and thus to the goal of a modular, maintainable and testable code base.

The Sprat PDE solver DSL is focused on the implementation of special-purpose finite-element method (FEM) solvers. Thus, it targets developers of such algorithms rather than FEM practitioners. That is why it does not feature the most abstract concepts of FEMs (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 the listing in Fig. 3:

1. Lazily evaluated matrix-vector arithmetic with a natural and declarative syntax. E.g., in Line 13, the evaluation of the right hand side and the assignment to the element vector `F_L[tau]` for mesh element `tau` is computed using only a single loop and without creating any temporaries.

2. Iterations over sets. One of the most common tasks found in numerical algorithms is to use an integer variable to iterate over some index range. The drawback of this approach is that the iteration variable has no semantic connection with the objects that is iterated over. Because of that, we added iterations over sets which explicitly state that, e.g., with the variable `tau` we iterate over all the elements of our mesh. Moreover, as the iteration variables are thin wrappers around indices, functionality related to the object they represent can directly be requested from them (for example in Line 10, we ask the element degree of freedom (DoF) `i` for its global index).

3. Single program multiple data (SPMD) abstractions for parallelization. We use MPI[2] to distribute the computations across many compute nodes. Many data types, such as the `DistributedVector` or the mesh classes, feature high-level abstractions to transparently handle data exchange between nodes. For example, in Line 16 the distributed vector `u` is instructed to exchange data regarding duplicated DoF in the mesh after being updated in Line 15. The method for calculating the dot product of `u` and `q` in Line 15 is also aware of the distributed nature of the problem and automatically computes the right value for the *global* problem and not just the node-local answer. To achieve good data locality, we combine MPI with OpenMP.[3] OpenMP is used to automatically execute vector operations, such as the update of `u` in Line 15, in parallel. Additionally, we introduce our own macro (`foreach_omp` in Line 6) to use range-based for loops with OpenMP.

The most important feature to make the PDE solver DSL convey the impression of an independent *language* – rather than a *library* –, is the natural syntax for matrix-vector arithmetic. To implement this feature, we rely on template meta-programming techniques combined with operator overloading. The template capabilities of C++ allow to construct abstract syntax trees in the form of template types.

[2]http://www.mpi-forum.org
[3]http://www.openmp.org

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

Figure 3: Sprat PDE solver DSL code snippet

E.g., a binary expression node can be represented by a type that is templated with its two child nodes. Because the implementation of this is tedious and error-prone (esp. regarding the type system and transformations of the abstract syntax tree (AST)), we use Boost Proto [13], which is itself a DSL for embedding DSLs into C++. Boost Proto provides means for constructing, transforming, and executing template expression in the form of an AST. It allows specifying a grammar for a DSL and automatically takes care of all the necessary operator overloadings.

Embedding a DSLs into C++ this way offers the great advantage of getting full language support without any additional effort. There are, however, two drawbacks to this approach. First, the generation step from the DSL to the target language is implicit and thus there is no generated code that could be inspected. This can partly be overcome by looking at the output of different compiler stages, although we recognize that this can hardly compete with well-formatted code from an explicit generation step. A second drawback is concerned with error reporting. It is well known that many C++ compilers generate long and complicated error messages when it comes to errors concerning templated types. But even if this was overcome, the error reporting would not be on the level of abstraction on which the users write their code in the DSL (i.e., matrices and vectors and not template data types; cf. [12]). To mitigate this problem, the author of a C++ DSL should make use of static assertions which are checked at compile time to report errors on the level of abstraction of the model.

The Sprat PDE solver DSL is very concise and it allows to express mesh-based PDE algorithms in a way that closely resembles their representation in mathematical text books or articles. Because of this high abstraction level, it is relatively easy to make typical changes to the algorithm (such as generalizing it to more dimensions) and to check whether the implemented algorithm actually corresponds to the algorithm in some formal description (as they almost look the same). The compact notation for matrix-vector arithmetic also encourages users to write tests because checking, for example, whether the assertion holds that all entries of a vector sum are positive is just a one-line statement (`assert(u+v > 0)`) instead of a loop over their indices etc. Furthermore,

it can be assumed that the language is easy to learn for a numerical mathematician. The user has to use only a handful of classes plus the very natural syntax for matrix-vector arithmetic and iterations over sets. Additionally, the structure of FEM algorithms is usually very similar, which makes it possible to supply the user with a skeleton for the implementation. Such a skeleton also encourages users to employ all the features of the DSL rather than implementing existing features again in the host language.

Concerning the maintenance of the PDE solver DSL, users would most likely want to add new data types, such as special-purpose matrix formats or mesh types. For a language that is as closely embedded into its host language as the PDE solver DSL this will likely prove to be difficult: while Boost Proto simplifies the process of introducing new types, it is still far from trivial to correctly implement all of their interactions with other data types. However, we do not consider this a serious concern since it is very unlikely that the provided set of matrix types is inadequate and, apart from that, new mesh types can be introduced by filling in the gaps of a class skeleton.

4.2 The Sprat Ecosystem DSL

In order to apply the fish stock model implemented with the Sprat PDE solver DSL, it has to be parameterized for a specific marine ecosystem by a stock assessment scientist. For this purpose, we designed 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. The unparameterized model already lends itself towards this level of abstraction and, thus, constitutes a semantically rich domain-specific platform for the ecosystem DSL.

Figure 4 shows an example of a simulation description using the Sprat ecosystem DSL in the corresponding Eclipse editor view 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. Unit support is vital for such a description language as there are numerous popular examples of mission-critical failures resulting from unit inconsistencies in numerical software (e.g., the Mars Climate Orbiter crash due to the mixed use of non-/metric units [10]). 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 (which would be kilograms in the depicted case). Unit conversions (e.g., from degree Fahrenheit to degree Celsius) are automatically carried out by the DSL. As some attributes might be specified in relation to another quantity (e.g., a growth coefficient is specified for a certain temperature), a modifier can be introduced to these attributes with the `@` keyword. Another keyword of the language is `record`. It 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 huge amount of all data generated by the simulation. Within the record expressions, there are special functions to refer to model data. These function are called using named parameters for better readability and the arguments can be intervals (`from ~ to`) with optional endpoints.

As can be seen from the example in Fig. 4, the structure of the DSL is straightforward and only contains concepts that the targeted domain experts should be familiar with. Nonetheless, it is critical for the acceptance of any DSL to guide users while they produce artifacts in the language. To this end, the editor offers a list of content proposals at any given position in the document. Not only do these context-sensitive suggestions include singular items, such as keywords, functions, and units, but also complete templates for, say, a new species entity and all of its necessary attributes. An example of this feature is displayed at the bottom of Fig. 4, where the name of the last missing species attribute (and only this missing one!) is proposed. As long as not all necessary attributes are specified, meaningful error messages are raised in appropriate locations.

Beyond model completeness, the validator of the DSL checks various other constraints to assure that the description of the simulation is sound and will result in a successful and meaningful simulation run. Especially on the higher levels of the DSL hierarchy, it is important to make sure that all errors are detected before generating into the next lower level, since, during this process, abstraction will be lost. Thus, it would no longer be possible to communicate problems to the domain experts on the level of abstraction that they are familiar with and that they implemented their model in. Alternatively, one could introduce measures to automatically reconstruct the higher-level concepts from the generated code. This, however, would require an additional development effort and it is likely to be beyond the expertise and interest of the intended user community.

Regarding the technical implementation of the Sprat ecosystem DSL with Xtext, we already mentioned that the Java-based framework generates a lot of default functionality and code stubs. All the functional components of the DSL runtime reside in their own module and are composed using dependency injection. This makes it possible to overwrite and customize nearly all aspects of the language runtime in a modular way. For our purpose, this is especially interesting in the context of the generator as it allows us to easily switch between different generator implementations at runtime to target different fish stock models with exactly the same simulation description.

The generator module that we implemented for our DSL hierarchy produces C++ code. We took care to separate the generated code from user-written code because generated files are just overwritten without warning by our generator. In C++ and some other object-oriented target languages, one option to achieve this separation is the generation gap pattern that works with inheritance [6]. But even though the generated code is not meant to be consulted by the developer on the lower hierarchy layer, we made sure that it is well-formatted and even tried to preserve some of the abstractions of the higher layer, for example by stating the unit of every attribute in the generated code in a comment. This way, we maximize clarity and help to prevent problems with different interpretations that can appear at the transition of two DSL hierarchy levels due to the model transformation.

While Xtext encourages the creation of a DSL runtime infrastructure comprised of loosely coupled modules, we introduced a single central configuration class used by all these modules. In this configuration class, we describe all the attributes, units, functions etc. that belong to the language. The Java code (cf. Fig. 5) is as declarative as possible and

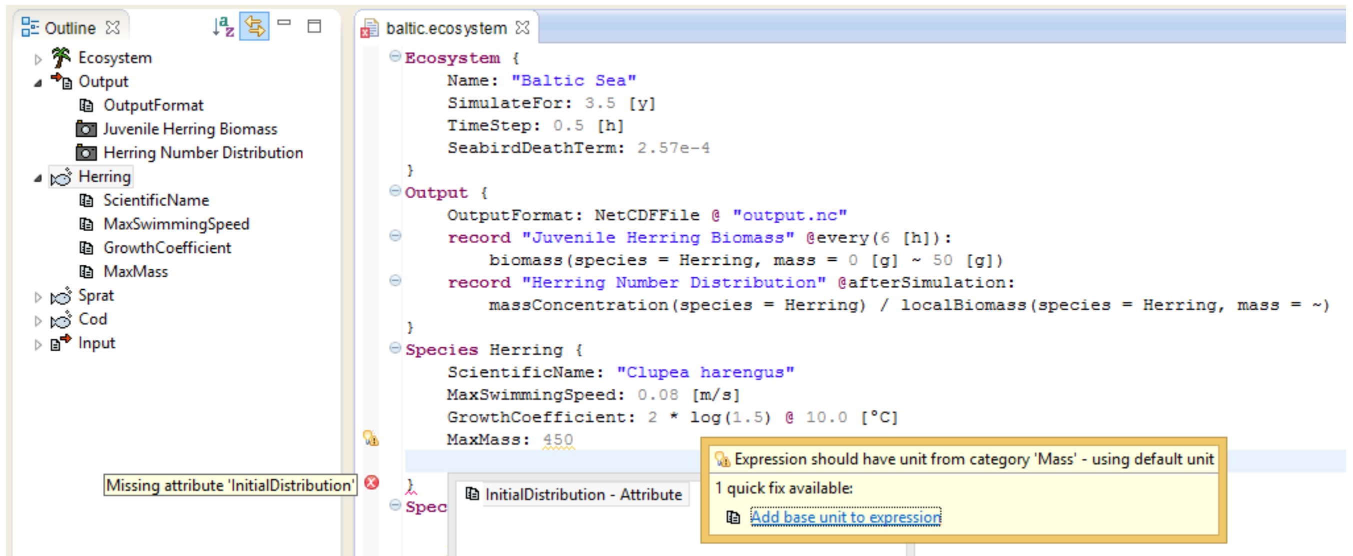

Figure 4: Eclipse editor for the Sprat ecosystem DSL

```
MASS.add(new SpratUnit("kg", 1.0    ));
MASS.add(new SpratUnit("g" , 1.0e-3));
SPECIES_ATTRIBUTES.add(
  new SpratAttribute("MaxMass", MASS,
    new ValueRange(0.0, ValueRange.INFINITY)));
```

Figure 5: Java snippet for configuring the Sprat Ecosystem DSL runtime

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

Figure 6: Ansible task for configuring OpenStack instances

focuses on readability for non-developers. This allows the users of the ecosystem DSL (who are likely not trained as software developers) to modify the ecosystem DSL themselves at least to some extent by simply copying, pasting, and customizing some code fragments. This design with a central configuration class makes it possible to introduce a new unit or attribute in all modules such as the generator, the content assist etc. by merely adding a single line of code to one file.

With its high-level abstractions and its seamless integration with the lower layers of the DSL hierarchy, the Sprat ecosystem DSL facilitates the collaboration between experts from different disciplines. Its concise and declarative syntax ensures that the resulting models are descriptive and maintainable. The intuitive language design and the full tool support minimize hurdles in adopting the DSL.

4.3 The Deployment DSL – Ansible

Once a fully parameterized model is implemented by the three lower layers of the DSL hierarchy, a last piece of information is missing for a complete executable simulation: a description is needed of how to deploy it in a specific distributed computing environment. As for this purpose there exist multiple automation tools, each with their own deployment DSL, we choose to use an existing solution rather than implementing our own DSL for the uppermost level of the DSL hierarchy.

The decision of whether or not to use an existing DSL for a specific layer of a DSL hierarchy in the Sprat approach, is a trade-off that includes user-acceptance and implementation effort. Using an existing language should result in less implementation effort as the DSL only has to be integrated but does not need to be developed from scratch. However, with a newly developed DSL one has full control over the syntax and can thus tailor the language to the specific aspect of the domain it is used for. Ultimately, the decision of whether to use an existing DSL depends on the complexity of the language: for complex languages, the development effort would be high and the targeted domain experts would have to put in some work in order to become proficient in the newly developed DSL. In such cases, one would opt for an existing quasi-standard DSL in that domain, if such a language exists. For simpler languages, both the development process is relatively easy thanks to language workbenches like Xtext and such languages are likely to be straightforward to learn by the targeted audience.

From the range of existing automation tools (cf. Sect. 5.3), we selected Ansible and its YAML-based Playbook DSL. Ansible playbooks are state-driven insofar as they describe desired configurations of systems rather than just actions (cf. Fig. 6). Additionally, they can contain template expressions for code generation.

The modular structure of playbooks allows us to separate the phase of configuring the environment from the phase of deploying and running the simulation. This way, we can independently implement configuration modules for different computing environments from bare-metal clusters to different private or public cloud computing providers. Specifi-

cally, we implemented such a module for a private cloud based on OpenStack.[4] This module makes sure that a user-configurable amount of compute nodes with a suitable environment is running and then uploads the simulation data as well as a second playbook to the master node. This second playbook takes care of configuring the compute nodes from within the cloud and thus only has to know the addresses of the other nodes but, apart from that, can be completely agnostic of the concrete infrastructure it is deployed to.

The Ansible playbook language as well as the configuration description we realized with it fill the last gap of the simulation that is implemented by using the DSL hierarchy in the Sprat approach. The fact that playbooks describe states rather than actions results in a compact and descriptive deployment specification. The modular structure of our implementation makes this specification well-maintainable and allows to adapt it to changing requirements (such as deploying to a public cloud) without great effort.

4.4 Combining Internal and External DSLs for Apt Domain-Spanning Abstractions

Our experience with applying the Sprat approach to the evaluation example suggests that it is beneficial to use internal DSLs on lower layers of the language hierarchy and external DSLs on higher levels to achieve an implementation that is both descriptive across domain borders and performant.

In the context of scientific software development – which is targeted by the Sprat approach –, the lower layers of the DSL hierarchy will usually be responsible for implementing numerical modules. Here, it is important to be able to implement problem-specific data structures and to achieve good runtime performance. Internal DSLs embedded into GPLs that utilize abstractions close to the underlying hardware are a good fit for this requirement. Benchmark experiments – which we cannot discuss here in detail because of space constraints – show that the abstractions introduced with our Sprat PDE solver DSL worsen runtime performance by less than 5 % compared to a reference implementation. With domain-specific optimizations, e.g. related to similar sparsity patterns of matrices, the PDE solver DSL with its higher-level abstractions becomes even slightly faster than the named reference implementation.

The higher layers of the DSL hierarchy are usually concerned with abstractions from domains that are less closely related to computer science and programming in general. Here, the flexible and concise syntax of external DSLs with good tool support is adequate to help domain experts to focus on expressing their solutions and not on programming language syntax.

In the Sprat approach we promote to generally favor the use of a different DSL for each layer of the hierarchy instead of, e.g., one DSL that features the different layers as viewpoints. In this way, we maximize the flexibility of the approach with regard to assigning persons to roles and concerning the separation of concerns. It is likely that the development of a new scientific software will start with fewer persons than the number of disciplines and, thus, roles involved. Hence, a single developer will have to take on different roles in the Sprat approach. If new developers join the team, they can take over a certain role related to their scientific discipline and start working with a DSL that does not interact with any other (possibly unfamiliar) discipline. Furthermore, this approach – as opposed to the viewpoint approach – makes it possible to exchange a single DSL in the hierarchy with a new language (e.g., if a new (quasi-)standard DSL emerges for a field). In this case, only the generator of the supraordinate DSL would have to be adapted by the role of the software language engineer.

For our evaluation example this implies that, for example, a single person (fulfilling the three roles of the software language engineer, the numerical mathematician, and the ecological modeler) could implement the DSLs, the solver, as well as the unparameterized version of the fish stock model and then distribute it to various stock assessment scientists. Each of them can complete the model in a different way without any knowledge of any other layer and discipline involved.

The Sprat approach is one example of how to adapt established methods of software engineering in order to successfully transfer them to the domain of scientific computing. Our evaluation example shows that Sprat and its hierarchical combination of internal and external DSLs facilitates the cooperation of scientists from different disciplines without the necessity of turning scientists into software engineers.

5. RELATED WORK

Among the few that consider a whole MDSD approach for scientific computing and not just the design of a single DSL are Palyart et al. [14]. They, however, concentrate on abstracting from different hardware platforms and disregard the aspect of collaboration in interdisciplinary teams. As their approach is based on OMG MDA, they present only a single DSL that focuses on model transformations rather than on direct code generation.

Below, we discuss related work on hierarchies of DSLs, PDE solver DSLs, and DSLs for deployment.

5.1 Hierarchies of DSLs

In his overview of design patterns for DSLs, Spinellis [17] describes the pipeline pattern to compose families of DSLs. In a DSL pipeline, each language handles its own syntax elements of an input model that is then passed on further down-stream. Our concept of DSL hierarchies extends this pattern by letting each lower layer act as a domain-specific platform for the supraordinate layer. Additionally, we augment this pattern with different roles to form the Sprat development approach.

Preschern et al. [16] as well as Prähofer and Hurnaus [15] 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 [16] or hierarchical components [15], respectively.

5.2 PDE Solver DSLs

Blitz++ [20], Eigen,[5] and Armadillo[6] 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

[4] `http://www.openstack.org`

[5] `http://eigen.tuxfamily.org`

[6] `http://arma.sourceforge.net`

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 provides a DSL (Finite Element Form Language) for specifying FEM discretizations and variational forms [1]. The level of abstraction of such a language is higher than that of our DSL and targets FEM practitioners rather than algorithm developers. Additionally, the FEniCS framework is limited to three-dimensional problems as are most similar tools (cf. the related work section of [1]).

Liszt [3] aims at the same level of abstraction 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.

5.3 DSLs for Deployment

Besides Ansible, the most popular tools for automated IT administration and deployment are Puppet[7] and Chef.[8] In contrast to Ansible, however, they feature "heavyweight" clients on the maintained machines and, by default, employ a pull rather than a push scheme for configuration changes.

Another alternative to Ansible is CodeCloud and its XML-based Cloud Job Description Language that can describe the deployment of compute jobs in the cloud [2]. It, however, focuses only on the cloud and not on bare-metal compute environments.

6. CONCLUSIONS

Our Sprat approach that is organized around the architectural design of a DSL hierarchy aims to facilitate the cooperation of scientific software developers from different fields and, at the same time, to improve the code quality of such projects. We demonstrated the capabilities of the approach by discussing its application to the implementation of a marine ecosystem simulation.

In the future, we plan to assess the adaptability of the Sprat approach to other areas of interdisciplinary scientific software development, such as climate modeling. It would also be of interest to apply the approach in domains beyond science such as in the automotive software domain and to compare it with existing collaboration approaches in this area, such as AUTOSAR.[9] Furthermore, we plan to evaluate the potential of DSLs that represent models graphically or semi-graphically (i.e., as structured text such as tables) in the context of the Sprat approach.

7. REFERENCES

[1] M. S. Alnæs. *UFL: a Finite Element Form Language*, volume 84 of *LNCSE*, pages 299–334. Springer, 2012.

[2] M. Caballer et al. CodeCloud: A platform to enable execution of programming models on the clouds. *Journal of Systems and Software*, 93:187–198, 2014.

[3] Z. DeVito et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings SC'11*. ACM, 2011.

[4] S. Easterbrook and T. Johns. Engineering the software for understanding climate change. *CiSE*, 11(6):65–74, 2009.

[5] S. Efftinge et al. Xbase: implementing domain-specific languages for Java. In *Proceedings GPCE'12*, pages 112–121. ACM, 2012.

[6] M. Fowler. *Domain-Specific Languages*. Pearson, 2010.

[7] J. Gray. eScience: A transformed scientific method. In *The Fourth Paradigm: Data-Intensive Scientific Discovery*, pages XVII–XXXI. Microsoft Research, 2009.

[8] J. Hannay et al. How do scientists develop and use scientific software? In *Proceeding SECSE'09*, pages 1–8. IEEE, 2009.

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

[10] J. Knight. Safety critical systems: challenges and directions. In *Proceedings ICSE'02*, pages 547–550. IEEE, 2002.

[11] T. Kosar et al. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, 17(3):276–304, 2012.

[12] M. Mernik et al. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

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

[14] M. Palyart et al. MDE4HPC: An approach for using model-driven engineering in high-performance computing. In *Proceedings SDL'11*, volume 7083 of *LNCS*, pages 247–261, 2012.

[15] H. Prähofer and D. Hurnaus. Monaco – a domain-specific language supporting hierarchical abstraction and verification of reactive control programs. In *Proceedings INDIN'10*, pages 908–914. IEEE, 2010.

[16] C. Preschern et al. Domain specific language architecture for automation systems: an industrial case study. In *Proceedings ECMFA'12*, 2012.

[17] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

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

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

[20] T. L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In *Advances in Software tools for scientific computing*, volume 10 of *LNCS*, pages 57–87. Springer, 2000.

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

[7] `http://puppetlabs.com`
[8] `http://www.getchef.com`
[9] `http://www.autosar.org`