A Domain-Specific Language and Editor for Parallel Particle Methods

SVEN KAROL, TOBIAS NETT, and JERONIMO CASTRILLON, Technische Universität Dresden, Faculty of Computer Science, Dresden, Germany
IVO F. SBALZARINI, Technische Universität Dresden, Faculty of Computer Science, Dresden, Germany and Center for Systems Biology Dresden, Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany

34

Domain-specific languages (DSLs) are of increasing importance in scientific high-performance computing to reduce development costs, raise the level of abstraction, and, thus, ease scientific programming. However, designing DSLs is not easy, as it requires knowledge of the application domain and experience in language engineering and compilers. Consequently, many DSLs follow a weak approach using macros or text generators, which lack many of the features that make a DSL comfortable for programmers. Some of these features—e.g., syntax highlighting, type inference, error reporting—are easily provided by language workbenches, which combine language engineering techniques and tools in a common ecosystem. In this article, we present the Parallel Particle-Mesh Environment (PPME), a DSL and development environment for numerical simulations based on particle methods and hybrid particle-mesh methods. PPME uses the Meta Programming System, a projectional language workbench. PPME is the successor of the Parallel Particle-Mesh Language, a Fortran-based DSL that uses conventional implementation strategies. We analyze and compare both languages and demonstrate how the programmer's experience is improved using static analyses and projectional editing, i.e., code-structure editing, constrained by syntax, as opposed to free-text editing. We present an explicit domain model for particle abstractions and the first formal type system for particle methods.

CCS Concepts: • **Software and its engineering → Application specific development environments**; • **Computing methodologies** → *Agent/discrete models*; *Simulation languages*; • **Mathematics of computing** → Solvers;

Additional Key Words and Phrases: Language workbenches, mathematical software, MPS, particle methods, scientific computing

ACM Reference format:
Sven Karol, Tobias Nett, Jeronimo Castrillon, and Ivo F. Sbalzarini. 2018. A Domain-Specific Language and Editor for Parallel Particle Methods. *ACM Trans. Math. Softw.* 44, 3, Article 34 (March 2018), 32 pages.
https://doi.org/10.1145/3175659

This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden."
Authors' addresses: S. Karol, T. Nett, and J. Castrillon, Chair for Compiler Construction, Georg-Schumann-Straße 7A, 01187 Dresden, Germany; emails: {Sven.Karol, tobias.nett, jeronimo.castrillon}@tu-dresden.de; I. F. Sbalzarini, Andreas-Pfitzmann-Bau (APB), room 2002, Nöthnitzer Str. 46, 01187 Dresden, Germany; email: ivos@mpi-cbg.de.

ACM 0098-3500/2018/03-ART34 $15.00
https://doi.org/10.1145/3175659

1 INTRODUCTION

The emergence of massively parallel hardware architectures, such as different kinds of multi- and many-cores, general-purpose GPUs, and FPGAs in scientific high-performance computing (HPC) has led to the development of new (or the renovation of old) programming models, paradigms, languages, and standards. Standardized interfaces, such as the Message Passing Interface (MPI) (The MPI Forum 2012), OpenMP (OpenMP Architecture Review Board 2013), OpenACC (OPENACC-STANDARD.ORG 2012), or hardware-specific low-level programming languages, such as CUDA (nvidia 2015), made their way into HPC programming as libraries, language extensions, or compilers. However, using these tools efficiently in scientific programming requires in-depth knowledge of the underlying HPC architecture, the development of parallel applications, and numerical simulation methods. Hence, the achievable level of abstraction remains rather low, which is a well-known problem in scientific programming (Hannay et al. 2009; Wilson 2006), causing the "knowledge gap" in program efficiency (Sbalzarini 2010).

To address this gap, scientific libraries and domain-specific languages (DSLs) have evolved into an important tool set in scientific HPC. However, most of the scientific DSLs are built on rather conventional technology, such as macros, templates, and/or parser generators. In recent years, more sophisticated tools have been proposed, frequently referred to as *language workbenches* (Fowler 2005; Erdweg et al. 2013), which enable developers to more easily create and integrate DSLs following a model-centric approach. A main driver behind the rising interest in such tools is the paradigm of *language-oriented programming* (Ward 1994), where DSLs are created to describe and solve software problems instead of using general-purpose languages, with the goal of increased productivity and better maintainability through abstraction. Models are the central paradigm that is edited by users and automatically transformed or interpreted by the workbench tooling. From this integrative idea, major advantages over conventional approaches arise. Most language workbenches provide configurable features known from professional programming environments, such as automatic code completion, refactoring, and syntax highlighting. Moreover, they typically provide a collection of internal, tailor-made specification languages that address common concerns in language development, e.g., languages for pretty-printing, rewriting, parsing, and code analysis or generation.

These tools were not used when designing the Parallel Particle Mesh library (PPM) and the Parallel Particle Mesh Language (PPML) as a library and a DSL for large-scale scientific HPC using particle-mesh abstractions (Sbalzarini et al. 2006; Sbalzarini 2010; Awile 2013; Awile et al. 2013). Instead, PPML was implemented conventionally as an *internal DSL*, embedded into Fortran 2003. However, as PPML does not have a completely integrated language model, it is difficult to maintain, debug, extend, or optimize PPML programs (Karol et al. 2015). To improve on these issues, we developed the PPM Environment (PPME) as an Integrated Development Environment (IDE) for particle-mesh methods. Based on the Meta Programming System (MPS) (Dmitriev 2004; MPS - 3.2 - Documentation 2015b), a language workbench that closely follows the ideas of language-oriented programming, PPME provides an additional layer of abstraction on top of the PPML stack (cf. Figure 11 in Section 5). In contrast to text-based language workbenches, MPS relies on *projectional editing* where users directly operate on a rendered, form-like "projection" of the program (Feiler and Medina-Mora 1981). This enables advanced rendering of tables and mathematical equations inlined with normal program code. Due to its underlying principles, we believe that MPS is an excellent platform to design languages that address the "knowledge gap" and raise the level of abstraction in scientific programming.

In this article, we present PPME as the first IDE for high-performance particle simulations. We introduce a complete language model that provides the corresponding abstractions and paves the way for further domain-specific analyses. As an important first analysis, we implement a static

```
class PARTICLE {
    vector(space-dimension) :: position x⃗, positionChange Δx⃗
    struct :: properties ω⃗, propertiesChange Δω⃗

    method [vector K⃗_x, struct K⃗_ω] interact(PARTICLE q)
    method evolve()
}
```

Fig. 1. Particle declaration and properties in pseudo code.

type-inference engine, supported by a formal type system. Furthermore, we demonstrate the advantages of this approach and of using language workbenches for numerical optimizations by integrating a mechanism for error-reduction in floating-point expressions.

The remainder of this article is structured as follows: Section 2 briefly introduces the background of particle methods. Section 3 provides a more detailed overview and analysis of the current PPML implementation and tool flow. The language model and type system of PPME are discussed in Section 4. Section 5 gives an overview of PPME's coarse-grained architecture and implementation and presents three case studies. In Section 6, we show how optimizations can be added by integrating an external analysis tool using domain-specific information. A qualitative evaluation of our work is given in Section 7. Finally, we discuss related work in Section 8, and Section 9 concludes the article.

2 PARTICLE METHODS

Particle methods provide a universal approach for numerical simulations in scientific computing. In contrast to other simulation frameworks, such as finite element methods (FEM) or Monte Carlo methods, particle methods can simulate models of all four kinds: discrete/deterministic, discrete/stochastic, continuous/deterministic, continuous/stochastic (Sbalzarini 2013). In the case of continuous models, particles correspond to discretization points. When discrete models are simulated, entities in a model are directly represented by particles. In deterministic simulations, particle positions and properties evolve according to deterministic interactions between particles, whereas in stochastic models, these interactions are probabilistic.

In general, particles are zero-dimensional point-like objects characterized by a collection of properties of arbitrary types and a position in any space given as a vector whose length corresponds to the dimension of that space. While a particle always has a position, its list of properties may grow or shrink in the course of a simulation. As an example of a discrete particle, consider a car on a street. The car's position may correspond to its GPS coordinates on a map while its properties could be velocity, the driver's age, number of passengers, or the color of the car. Other examples may be a pixel of an image (i.e., in a discrete space) or a discretization point of a continuous mathematical field, where the space is continuous.

Particles can *interact* pairwise with other particles and they can *evolve*. Evolving means that a particle's position and/or properties change due to its own state and/or the states of other particles in the domain. The influence of other particles is due to the interactions, which may yield a contribution to the change. Hence, in pseudo code, the essential ingredients of particle objects can be described as shown in Figure 1. Using this very basic interface, the evolution of a particle may depend on the position vector $\vec{x}$ and the list of properties $\vec{\omega}$ of the particle itself, as well as the values of all other particles in the system. If we assume that all particles influence each other, in the most general form, then the changes for any particle p in the system can be described abstractly by Equation (1):

$$\begin{bmatrix} \Delta\vec{x}_p \\ \Delta\vec{\omega}_p \end{bmatrix} = \sum_{q=1}^{N} \begin{bmatrix} \vec{K}_x \\ \vec{K}_\omega \end{bmatrix} = \sum_{q=1}^{N} \vec{K}(\vec{x}_p, \vec{x}_q, \vec{\omega}_p, \vec{\omega}_q)\,. \tag{1}$$

Here, N is the total number of particles in the system, which may change over time (e.g., depending on a boundary condition). $\vec{K}$ represents the *interaction kernel* that encapsulates the computational model and corresponds to a mathematical representation of the `interact` method. Applied on position vectors $\vec{x}_p, \vec{x}_q$ and properties $\vec{\omega}_p, \vec{\omega}_q$, the kernel produces the elementary changes $\vec{K}_x$ and $\vec{K}_\omega$ of the pairwise interactions between particles p and q. The cumulative change for particle p due to all interactions with other particles is represented by two deltas, $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$, which are used by `evolve` to update the property values and position of p.

In numerical simulations, updates of particle properties and positions occur at each time step. Thus, it is important to evaluate the pairwise interactions efficiently. In the worst case, each particle interacts with each other particle, which leads to quadratic time complexity. However, typically, a particle only needs to interact with its "neighbors" within a finite range. In such cases, optimized data structures such as cell lists (Hockney and Eastwood 1988) exist, which allow for computing particle interactions in linear time (average complexity if particles are uniformly distributed). Nevertheless, the worst-case complexity remains quadratic if all particles are located within the interaction range or the interaction range is the size of the domain. In these cases, efficient approximation algorithms are available, e.g., the Barnes-Hut algorithm (Barnes and Hut 1986) and Fast Multipole Methods (Greengard and Rokhlin 1987). Another way to address this problem is to use a hybrid particle-mesh approach, where interactions of finite range are computed using particles, whereas interactions of infinite range are evaluated using mesh-based approaches (Hockney and Eastwood 1988).

The range of the particle–particle interactions is defined by the support of the interaction kernel $\vec{K}$. This kernel is the mathematical representation of the system to be simulated and encapsulates all application-specific details. When simulating discrete models, $\vec{K}$ corresponds to the pairwise interaction potential between the entities in the model, e.g., the inter-atomic force fields in a molecular-dynamics simulation. When simulating continuous models, such as partial differential equations (PDEs), $\vec{K}$ contains the discretized continuous or differential operators. In this case, the particles as discretization/colocation points at which the value of the continuous function is sampled.

Particle discretizations of differential operators in PDEs (i.e., the kernel $\vec{K}$) can be determined using a variety of classical approaches from numerical analysis (Lucy 1977; Liu et al. 1995; Belytschko et al. 1994; Lancaster and Salkauskas 1981; Broomhead and Lowe 1988; Degond and Mas-Gallic 1989; Eldredge et al. 2002) that are generic to arbitrary linear differential operators. They all have in common that the kernel $\vec{K}$ is pre-computed, usually analytically by hand, and then implemented in the discrete form. To free the scientific programmer from this analytical calculation, we here implement a method known as *Discretization-Corrected Particle Strength Exchange (DC-PSE)* (Schrader et al. 2010). DC-PSE is a general particle discretization framework where the discrete kernels are automatically computed at runtime. In addition, DC-PSE also shows superior stability and accuracy properties compared to other mesh-free discretization methods (Schrader et al. 2010; Reboux et al. 2012; Schrader et al. 2012; Bourantas et al. 2016).

3 THE PARALLEL PARTICLE MESH LANGUAGE PPML

The Parallel Particle-Mesh Language (PPML) (Awile 2013; Awile et al. 2013) provides domain-specific abstractions to ease the development of distributed-memory particle-mesh simulations with the PPM HPC library (Sbalzarini et al. 2006; Sbalzarini 2010; Awile et al. 2010). The language is smoothly embedded into Fortran2003—the implementation language of PPM. The major advantage of relying on PPML over using the PPM library directly is that complex library protocols and parallelization code are hidden from the user. Instead, PPML provides first-class concepts for

particle programming such as *particles*, *neighbor lists* (with optimized implementations in PPM), particle properties, such as *vector* and *scalar fields* and *differential-operator definitions*. Furthermore, particle-specific foreach loops are supported, as well as loops over discrete time steps. For high-performance parallelization, PPML supports distributed memory with message passing based on MPI. Several macro commands are provided to help handle the MPI setup, create topologies (i.e., decomposing the domain and assign subdomains to processes), distribute particles over these topologies, and exchange data at subdomain boundaries (cf. Sbalzarini et al. (2006)).

3.1 A Simple Application Example

To illustrate how parallel simulations can be implemented in PPML, we discuss an example of a Gray-Scott reaction-diffusion system, taken from the PPML article (Awile 2013). A Gray-Scott reaction-diffusion system describes the concentrations (in normalized dimensionless units) of two chemicals u and v that react with each other and diffuse (Gray and Scott 1983). The process can be described by a system of two partial differential equations that define the evolution of the chemicals' concentrations, u and v, over time:

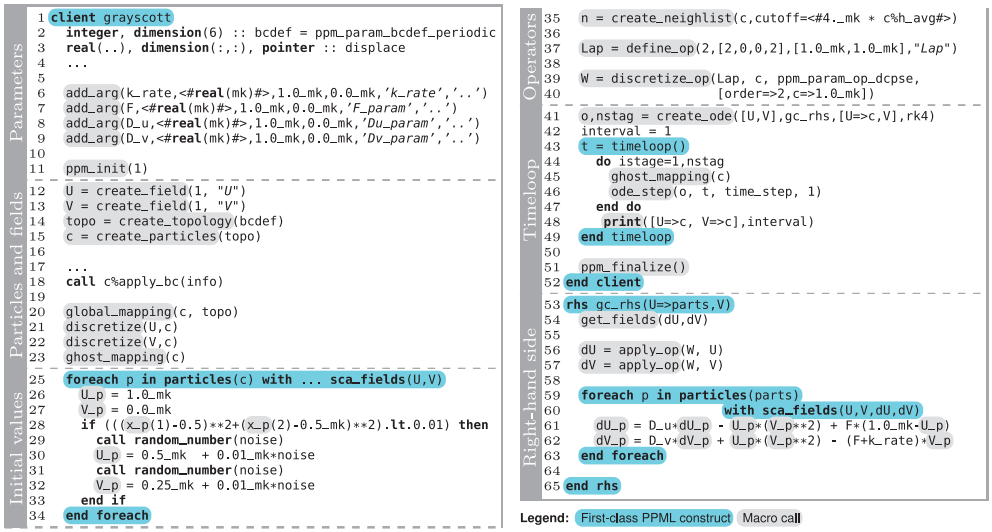
$$\frac{\partial u}{\partial t} = D_u \Delta u - uv^2 + F(1-u), \tag{2}$$

$$\frac{\partial v}{\partial t} = D_v \Delta v + uv^2 - (F+k)v. \tag{3}$$

Equation (2) describes the time derivative of u as a sum of three terms: First is the *diffusion term* $D_u \Delta u$, where D_u is a predefined diffusion constant and Δu the Laplacian (divergence of the gradient) of u. Second is the *reaction term* $-uv^2$ defining how much of u is converted to v by the reaction. The last term in Equation (2) is the *replenishment term*, defining how much of fresh u is added to keep the reaction alive, depending on a constant *feed rate* F. Equation (3) describes the time derivative of v also as a sum of reaction, diffusion and, instead of a replenishment term, a *diminishment term*. D_v is the constant diffusion rate of v and $-(F+k)v$ defines how much of v is taken out (consumed) from the system, depending on F and a *removal rate* k.

This continuous model can be solved numerically using particle methods and PPML. To do so, u and v are discretized as particle properties and particles are distributed over the entire domain. Furthermore, the differential operators (i.e., the Laplacians) need to be discretized according to the used method. After providing initial values of u and v at particles, an approximate solution can be computed for a series of time steps.

Figure 2 shows the corresponding PPML *client* program as a multi-part listing that highlights the different ingredients of the program. The first part on the left-hand side of the figure contains variable and constant declarations, the boundary condition, as well as declarations of external arguments allowing users to parametrize the simulation. The second part declares u and v as scalar fields `U` and `V` and discretizes them over particles. Furthermore, a topology is created to distribute the particles on a computer cluster. In the third part, the initial values of `U` and `V` are set using a PPML `foreach` loop that iterates over all particles in the domain, by default assigning `U` a value of 1 and `V` a value of 0. However, within a radius of $\sqrt{0.1}$ around the center, a small random amount of v is added to start the reaction. The fourth part (on top of the right-hand side of Figure 2) contains the definition and discretization of the Laplace operator and initializes a particle neighbor list with a specific `cutoff`. The cutoff (i.e., the range of the particle–particle interactions) is set such that each particle interacts with all neighboring particles that are closer than four times the average inter-particle distance. The remaining two portions of the figure specify the `timeloop`, which sequentially loops over the specified range of time steps, in each step evolving the solution by calling the PPM solver with the specified right-hand side, updating the particle properties, exchanging




```
1  client grayscott
2    integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
3    real(..), dimension(:,:), pointer :: displace
4    ...
5
6    add_arg(k_rate,<#real(mk)#>,1.0_mk,0.0_mk,'k_rate','..')
7    add_arg(F,<#real(mk)#>,1.0_mk,0.0_mk,'F_param','..')
8    add_arg(D_u,<#real(mk)#>,1.0_mk,0.0_mk,'Du_param','..')
9    add_arg(D_v,<#real(mk)#>,1.0_mk,0.0_mk,'Dv_param','..')
10
11   ppm_init(1)
12   U = create_field(1, "U")
13   V = create_field(1, "V")
14   topo = create_topology(bcdef)
15   c = create_particles(topo)
16
17   ...
18   call c%apply_bc(info)
19
20   global_mapping(c, topo)
21   discretize(U,c)
22   discretize(V,c)
23   ghost_mapping(c)
25   foreach p in particles(c) with ... sca_fields(U,V)
26     U_p = 1.0_mk
27     V_p = 0.0_mk
28     if (((x_p(1)-0.5)**2+(x_p(2)-0.5_mk)**2).lt.0.01) then
29       call random_number(noise)
30       U_p = 0.5_mk  + 0.01_mk*noise
31       call random_number(noise)
32       V_p = 0.25_mk + 0.01_mk*noise
33     end if
34   end foreach
35   n = create_neighlist(c,cutoff=<#4._mk * c%h_avg#>)
36
37   Lap = define_op(2,[2,0,0,2],[1.0_mk,1.0_mk],"Lap")
38
39   W = discretize_op(Lap, c, ppm_param_op_dcpse,
40                     [order=>2,c=>1.0_mk])
41   o,nstag = create_ode([U,V],gc_rhs,[U=>c,V],rk4)
42   interval = 1
43   t = timeloop()
44     do istage=1,nstag
45       ghost_mapping(c)
46       ode_step(o, t, time_step, 1)
47     end do
48     print([U=>c, V=>c],interval)
49   end timeloop
50
51   ppm_finalize()
52 end client
53 rhs gc_rhs(U=>parts,V)
54   get_fields(dU,dV)
55
56   dU = apply_op(W, U)
57   dV = apply_op(W, V)
58
59   foreach p in particles(parts)
60                          with sca_fields(U,V,dU,dV)
61     dU_p = D_u*dU_p - U_p*(V_p**2) + F*(1.0_mk-U_p)
62     dV_p = D_v*dV_p + U_p*(V_p**2) - (F+k_rate)*V_p
63   end foreach
64
65 end rhs
```

Fig. 2. PPML program to numerically solve the 2D Gray-Scott reaction-diffusion system on distributed-memory computer systems.

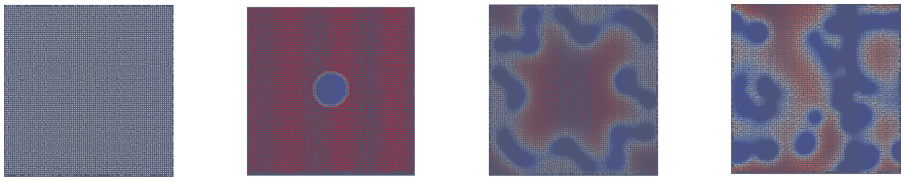


Fig. 3. Some intermediate results produced by the PPML Gray-Scott program.

data at inter-process boundaries, and printing the intermediate results to the file system. The right-hand side specification contains the reaction-diffusion equations to be solved with an explicit invocation of the discretized Laplacian over both fields yielding respective vectors of intermediate results. Another PPML particle `foreach` loop computes the contributions of each individual particle using LaTeX-like formula expressions, where underscores access individual particles. Figure 3 shows some intermediate results over a small domain at different time steps that have been produced by this PPML program, choosing $k = 0.051$, $F = 0.015$, $D_u = 2 \cdot 10^{-5}$, and $D_v = 10^{-5}$. For these parameters, the Gray-Scott system forms spatial patterns that are hypothesized since Alan Turing to be the chemical basis of biological growth and morphogenesis (Turing 1952).

3.2 Advantages Over Conventional Programming

The program shown in Figure 2 nicely demonstrates some of the major benefits of DSLs in scientific HPC. Most of the boilerplate code for instantiating PPM and managing parallelism with MPI is hidden from the developer. It is automatically generated by the PPML source-to-source compiler, emitting a plain Fortran program, which is then compiled and linked with the PPM library by a standard Fortran compiler. The size ratio of the PPML source and the generated Fortran program is 85:668, which means that the developer is freed from the burden of writing an extra 583 lines of boilerplate code.

The improved program readability is a further advantage of PPML over writing a plain Fortran program. Thanks to built-in domain-specific concepts and other specialized constructs, such as particle loops and underscore accessors, the program is more declarative and thus more readable, so that other domain experts can easily understand it. Finally, PPML was designed as an extensible language, apparently embedded into Fortran as a host language. This way, it circumvents one of the major obstacles of using closed, stand-alone DSLs, namely a lack of expressiveness that may prevent facets of a problem to be described using the abstractions at hand. If a problem cannot be described properly using PPML, then developers can always use plain Fortran (e.g., lines 28, 29, and 31) or they may define additional PPML macros. However, since PPML lacks a well-defined interface between the DSL and its host language, it is difficult to properly analyze the code and derive context information from it. This lack of context largely prevents automatic compile-time code optimization in PPML. Such optimizations are easier with non-embedded DSLs, where language interfaces need not to be considered.

3.3 Limitations in the Current PPML Design

The current design and implementation of PPML has some limitations that hamper code optimization and debugging. The most important limitation is that the language is not based on a formal domain model, which would enable reasoning about PPML programs to automatically check consistency, e.g., using a formal type system. Moreover, the language syntax is underspecified. Similar to an island grammar (Moonen 2001), only some parts of the language are modeled explicitly, while others remain undefined. Consider Figure 2 again. Parts of the program that are recognized by the PPML source-to-source compiler are highlighted in either gray (macro calls including list of arguments) or blue (first-class language constructs). These are the structural "islands" in the sense of an island grammar, while the non-highlighted parts are "water," i.e., parts of the program that are treated as a list of characters that do not provide any additional information to the PPML compiler. This fragmentary view on the code allows for only shallow analyses of input programs during the preprocessing phase, leaving most syntax and type errors undetected so that these are inherited by the generated program. If such a program is then fed into a Fortran compiler to produce an executable, then the compiler will detect these issues and associate them with the preprocessed code. However, the developer has neither seen nor written this automatically generated Fortran code and can therefore not trace back the errors to his PPML program. Debugging PPML programs is therefore unpractical. Even worse, some problems manifest only during or after execution by causing unintended results, e.g., through an unsuitable argument or wrong arrangement of calls.

By leveraging domain knowledge, one can define a complete domain model for the language implementation so that syntactic elements and semantic relations can be identified and used in a compiler. This way, syntactic problems and problems related to the language semantics can be detected early when processing DSL code (e.g., differential operators that are not used in an equation or have the wrong type of operand). This further provides the groundwork for improving user experience by adding features known from integrated development environments (IDEs) (e.g., syntax highlighting and code completion) and from compilers (e.g., optimizations such as tiling, program variants, or expression rewrites that only become possible through the additional information). This is enabled here by formulating a domain metamodel and a formal type system for the application domain of particle methods.

4 A DOMAIN METAMODEL AND TYPE SYSTEM FOR PARTICLE METHODS

A crucial step to overcome the current shortcomings of PPML is to develop a metamodel that structurally represents the domain of particle methods. This "domain model" enables compile-time

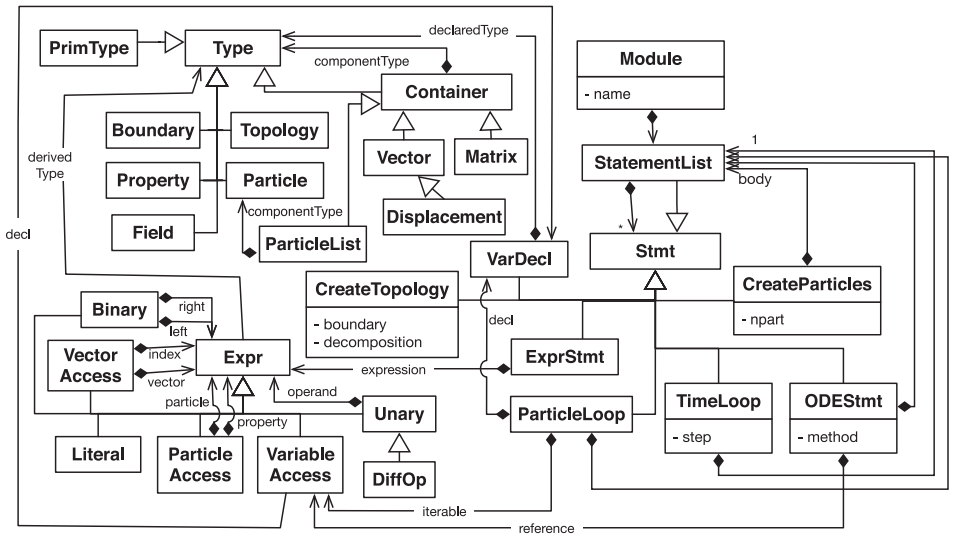


Fig. 4. A metamodel to describe the domain of particle methods.

reasoning by providing a structural basis for developing PPM programs. In this section, we describe such a model for particle methods, resulting in a static type system.

4.1 Domain Metamodel

The current PPML only supports a small fraction of the concepts that constitute a complete model. For instance, it provides constructs for defining computations over properties of particles in a domain, as well as loops for defining numerical simulations over a series of discrete time steps. However, since these concepts are only specified partially in PPML, it is not possible to reason about the actual computation steps.

In contrast to PPML, we propose a complete metamodel for particle methods that captures particles, particle data, and computations over these data. More precisely, it provides a means to specify particle data structures and properties. It supports high-level statements that group transformations of particle data, which allow modifying data within a certain scope. Furthermore, the model includes mathematical expressions to describe the actual governing equations. Finally, our metamodel defines the set and structure of potential data types supported in PPME.

Figure 4 shows an excerpt of the model as a UML class diagram. We derived its major ingredients from our experience of developing HPC applications with PPML. A single top-level concept `Module` contains a list of statements that describe a particle-based simulation. A statement, for instance, can be a composite `Timeloop` or an elementary command for creating a topology. Single computation steps are described using expression statements (`ExprStmt`). Expressions are binary or unary arithmetic and logical expressions that access constants, particle properties, and collections of particles. Furthermore, expressions can define differential operators (`DiffOps`) evaluated over particle properties, e.g., the Laplacian. Our model also provides a simple `Type` hierarchy with a set of primitive types (e.g., integers) and built-in domain-specific types. The domain-specific types can be atomic (e.g., `Particle` or `Topology`) or container types such as `Vector` or `ParticleList`.

When considering the model in Figure 4, it is evident that it only provides structural information, since it is limited to *potential* relations between objects and types, such as inheritance, composition,

and reference. Hence, while an instance of this model (i.e., a concrete specification of a particle method) may be syntactically correct, it may not fulfill requirements that are not specified in the model. Properties that impose such additional constraints need to be formulated as supplementary rules that derive additional information or check consistency of a specification (Bürger et al. 2011). As an example of where additional information needs to be derived, consider the `decl` reference that associates an access of a variable with a corresponding declaration. This is required, because users would not "draw" the corresponding connection but just "use" the variable via its name. Another important analysis computes the types of expressions, as, for instance, represented by the `derivedType` reference in the model, which associates an expression with a specific type. The rules that define this analysis can be captured conveniently using a formal type system (Plotkin 1981).

4.2 Types and Dimensions

Based on the domain model, we present a static type-inference mechanism, which relies on a formal type system for particle abstractions. The error detection capabilities resulting from the hierarchy of types and inference rules are key to constructively improve code quality of simulations written in PPME, as it detects errors at compile time and provides meaningful feedback to the developer. In addition, we present an optional unit calculus extension to the type system. This can be used to perform automatic consistency checks of expressions.

4.2.1 Type Hierarchy. The type hierarchy is built around the metamodel shown in Figure 4, i.e., all types derive from `Type` as a common supertype. The type system can be divided into two parts: a base type system and a domain-specific extension.

The base type system consists of a set of primitive types $\mathcal{P} = \{String, Boolean, Real, Integer\}$ and type-inference rules over this set. Additionally, $\mathcal{C} = \{Vector\langle X\rangle, Matrix\langle X\rangle\}$ represents a set of container types for matrices (i.e., tensors of rank 2) and vectors (i.e., tensors of rank 1) with components of type X. The set of base types $\mathcal{T}_{Base} = \mathcal{P} \cup \mathcal{C}$ is composed of primitive types $\mathcal{P}$ and container types $\mathcal{C}$.

These basic types are complemented by domain-specific types for particle methods, i.e., types that represent particles, particle lists, and different kinds of particle properties. These are: $\mathcal{D} = \{Particle, ParticleList, Field, Property, Displacement\}$. Furthermore, the boundary of the simulation domain and the data-distribution topology of the underlying PPM framework are captured in the set $\mathcal{O} = \{Topology, Boundary\}$. The set $\mathcal{T}_{PPM} = \mathcal{D} \cup \mathcal{O}$ of domain-specific types is then composed of $\mathcal{D}$ and $\mathcal{O}$.

Finally, $\mathcal{T} = \mathcal{T}_{Base} \cup \mathcal{T}_{PPM}$ denotes the set of all types in PPME. Note that this way of constructing $\mathcal{T}$ indicates the flexibility of language implementations in modern language workbenches like MPS and language-oriented programming in general. This fundamental type hierarchy can be extended in the future, adding new domain-specific concepts.

4.2.2 Syntax of Expressions. In PPME the standard set of expressions well-known by programmers is extended by domain-specific operations and expressions tailored for the domain. Figure 5 presents the syntax of expressions in PPME as production rules of a context-free grammar. Note that some domain-specific expressions (e.g., differential operators) are only available in a specific context.

Basic Syntactic Sets. The basic syntactic sets in PPME are comprised mainly of *literals* for primitive types and *variables* (cf. Figure 6). Literals are typed in a natural way, e.g., integers have type *Integer* and decimals have type *Real*. More complex sets can be derived from the basic syntactic sets for variables and literals. The abstract syntax of these derived syntactic sets is given by the form of expressions in PPME.

⟨*expr*⟩ ::= ⟨*expr*⟩ ('&&' | '||') ⟨*expr*⟩
| ⟨*expr*⟩ ('==' | '!=') ⟨*expr*⟩
| ⟨*expr*⟩ ('<' | '>' | '<=' | '>=') ⟨*expr*⟩
| ⟨*expr*⟩ ('+' | '-') ⟨*expr*⟩
| ⟨*expr*⟩ ('∗' | '/' | '^') ⟨*expr*⟩
| ⟨*unaryExpr*⟩

⟨*unaryExpr*⟩ ::= '-' ⟨*unaryExpr*⟩
| '!' ⟨*unaryExpr*⟩
| '√' ⟨*unaryExpr*⟩
| ⟨*primaryExpr*⟩

⟨*varAccess*⟩ ::= Identifier

⟨*primaryExpr*⟩ ::= ⟨*literal*⟩
| '(' ⟨*expr*⟩ ')'
| ⟨*varAccess*⟩
| ⟨*particleAccess*⟩
| ⟨*arrayAccess*⟩

⟨*literal*⟩ ::= IntegerLiteral
| RealLiteral
| StringLiteral
| BooleanLiteral

⟨*particleAccess*⟩ ::= ⟨*expr*⟩ '→' Identifier

⟨*arrayAccess*⟩ ::= ⟨*expr*⟩ '[' ⟨*expr*⟩ ']'

Fig. 5. Syntax of expressions in PPME.

booleans	b	$b \in \mathbb{B} = \{true, false\}$
strings	s	e.g., $s =$ "PPME"
integers	n, m	$n, m \in \mathbb{N}$
reals	r	e.g., $r = 3.14$ or $r = 6.62\text{E}{-}34$
variables	v	$v \in Var = \{a, b, \ldots, x, x_2, \ldots\}$

Fig. 6. Basic syntactic sets and their notation.

Unary Expressions ($\ominus(e)$, with $\ominus \in \{-, !, \sqrt{}\}$). PPME supports three unary operations, the unary minus $-e$, the logical not $!e$, and the square root $\sqrt{e}$. Obviously, this definition alone allows for "nonesense" expressions such as taking the square root of a string. The remainder of this section therefore presents rules for well-formedness and type conclusion to prevent erroneous phrases.

Binary Expressions ($e_1 \otimes e_2$, where $\otimes \in \otimes_{arith} \cup \otimes_{logi} \cup \otimes_{rel}$). Various binary operations are supported. First, PPME allows for typical arithmetic operators $\otimes_{arith} = \{+, -, *, /, \hat{}\}$ Second, there are operators for the logical *and* and *or* ($\otimes_{logi} = \{\&\&, ||\}$). Third, the common relational operators are available ($\otimes_{rel} = \{==, !=, <, >, <=, >=\}$). As for unary operations, the type system will check well-formedness of binary expressions and decide on the resulting type.

Domain-specific Operations. A strength of PPME is that domain-specific operations are seamlessly integrated into the language. They allow for concise notation of mathematical concepts, preserving the expressiveness of the mathematical notation. Following the domain model, fields and particle properties are defined on particle lists, and the language offers the syntactic concept *particle list access* (PLA) to access these fields and properties. In a similar manner, the value of a field discretized over particles, or any other property of a specific particle, can be accessed via a *particle access* (PA) operation. Given a particle list ps, a particle p from this list, a field f and a property x both defined on ps, the access operations of field f and property x are represented by an arrow:

$$ps{\rightarrow}f, \qquad ps{\rightarrow}x, \qquad p{\rightarrow}f, \qquad p{\rightarrow}x.$$

Intuitively, the result of a PLA is the whole field or property over the particle list. Additionally, PPME allows the developer to access the default properties of a particle, e.g., its position: $p{\rightarrow}pos$. Finally, there are notations for differential operators in the context of right-hand side statements. Simulation developers can use these operators when simulating continuous models (e.g., PDEs), staying close to the mathematical notation. This includes, for example, the Laplacian ($\nabla^2 e$) used in the Gray-Scott reaction-diffusion example.

Access Operations ($v[i]$, $m[i][j]$). The language also offers means to access elements of array-like structures such as matrices and vectors. Access operations are denoted by square brackets

$$\textsc{Var}\ \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \qquad \textsc{VarDecl}\ \frac{}{\Gamma \vdash \tau\ x : \Gamma \cup \{x = \tau\}} \qquad \textsc{VarInit}\ \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash \tau\ x = e : \Gamma \cup \{x = \tau\}}$$

$$\textsc{Paren}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau} \qquad \textsc{Assign}\ \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash x = e : \tau}$$

$$\textsc{VecAcc}\ \frac{\Gamma \vdash v : \mathbb{V}\langle\tau\rangle \quad \Gamma \vdash i : \mathbb{Z} \quad i \geq 0}{\Gamma \vdash v[i] : \tau} \qquad \textsc{MatAcc}\ \frac{\Gamma \vdash m : \mathbb{M}\langle\tau\rangle \quad \Gamma \vdash i, j : \mathbb{Z} \quad i, j \geq 0}{\Gamma \vdash m[i][j] : \tau}$$

$$\textsc{PartScaAcc}\ \frac{\Gamma \vdash p : \mathbb{P} \quad \Gamma \vdash f : \mathcal{E}\langle\tau, 1\rangle}{\Gamma \vdash p{\rightarrow}f : \tau} \qquad \textsc{PartVecAcc}\ \frac{\Gamma \vdash p : \mathbb{P} \quad \Gamma \vdash f : \mathcal{E}\langle\tau, n\rangle, n \geq 2}{\Gamma \vdash p{\rightarrow}f : \mathbb{V}\langle\tau\rangle}$$

$$\textsc{Unary}\ \frac{\Gamma \vdash e : \tau \quad \tau_{\ominus}(\tau) \neq \bot}{\Gamma \vdash \ominus e : \tau_{\ominus}(\tau)} \qquad \textsc{BinLog}\ \frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : \mathbb{B}}{\Gamma \vdash e_1 \otimes_{log} e_2 : \mathbb{B}}$$

$$\textsc{BinRel}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_{\otimes}(\tau_1, \tau_2) \neq \bot}{\Gamma \vdash e_1 \otimes_{rel} e_2 : \mathbb{B}} \qquad \textsc{BinAri}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_{\otimes}(\tau_1, \tau_2) \neq \bot}{\Gamma \vdash e_1 \otimes_{arith} e_2 : \tau_{\otimes}(\tau_1, \tau_2)}$$

$$\textsc{ErrUnary}\ \frac{\Gamma \vdash e : \tau \quad \tau_{\ominus}(\tau) = \bot}{\Gamma \vdash \ominus(e) : \mathbb{E}} \qquad \textsc{ErrBin}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_{\otimes}(\tau_1, \tau_2) = \bot}{\Gamma \vdash e_1 \otimes e_2 : \mathbb{E}}$$

$$\ominus \in \{\neg, !, \surd\},\ \otimes_{arith} \in \{+, -, *, /, a^b\},\ \otimes_{log} \in \{\&\&, ||\},\ \otimes_{rel} \in \{==, !=, <, >, <=, >=\}$$
$$\mathbb{B} = \mathit{Boolean},\ \mathbb{Z} = \mathit{Integer},\ \mathbb{R} = \mathit{Real},\ \mathbb{P} = \mathit{Particle},\ \mathbb{E} = \mathit{Error}$$
$$\mathbb{V} = \mathit{Vector},\ \mathbb{M} = \mathit{Matrix},\ \mathcal{E} = \mathit{Field/Property},$$

Fig. 7. Type rules for expressions in PPME.

containing the index to access. Similarly, elements of non-scalar particle properties can be accessed using the same notation. Let ps be a particle list with a non-scalar field f, and $p \in ps$ a particle from ps, then $p{\rightarrow}f[i]$ denotes the access of the ith element of f on particle p.

4.2.3 Formal Type System. Every literal and variable in PPME has an associated type, and the type of derived expressions often depends on their arguments' types. The formal type system describes the conclusions that can be drawn from a PPME program over its types, by defining rules for well-formedness of typed expressions. For instance, the PLA operator can only be used on particle lists, which is ensured by the static type-inference mechanism. Overall, the type rules ensure that expressions behave as expected in the context of particle methods.

A *typing environment* Γ associates variable names x and types τ as a set of pairs $\langle x, \tau\rangle$, commonly written as $x : \tau$. A lookup of a variable's type is denoted by $\Gamma(x)$, where $\Gamma(x) = \tau$ if and only if the environment contains an entry for the variable $\langle x, \tau\rangle \in \Gamma$. Otherwise, $\Gamma(x)$ is undefined. We further define the subtype relationship: if T and S are types, then $T < S$ denotes that T is a (direct) *subtype* of S, and S is a *supertype* of T. $T <^* S$ is the reflexive transitive closure of $<$, that is, S can be reached from T in the type hierarchy. In the remainder, we use $\leq$ to refer to $<^*$. We follow the notation of Clément et al. (1986). That is, each type-inference rule defines the conclusion that can be drawn if all n premises hold:

$$\frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

As premises, we allow typings as well as other predicates, e.g., for specifying a subtype relation.

Figure 7 shows the type rules for expressions implemented in PPME. The type of a variable is given by the typing environment Γ (rule Var). A variable declaration adds a new entry to the typing environment (rules VarDecl and VarInit). Type rules for unary (Unary) and binary operations (Bin*) can be defined with a general scheme, where the derived type information depends

$\tau_{+\mid-}(\tau_1,\tau_2)$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{V}\langle X\rangle$	$\mathcal{E}\langle X,n\rangle$
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{V}\langle\uparrow(\mathbb{Z},X)\rangle$	$\mathcal{E}\langle\uparrow(\mathbb{Z},X),n\rangle$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{V}\langle\uparrow(\mathbb{R},X)\rangle$	$\mathcal{E}\langle\uparrow(\mathbb{R},X),n\rangle$
$\mathbb{V}\langle Y\rangle$	$\mathbb{V}\langle\uparrow(Y,\mathbb{Z})\rangle$	$\mathbb{V}\langle\uparrow(Y,\mathbb{R})\rangle$	$\mathbb{V}\langle\uparrow(Y,X)\rangle$	$\bot$
$\mathcal{E}\langle Y,m\rangle$	$\mathcal{E}\langle\uparrow(Y,\mathbb{Z}),m\rangle$	$\mathcal{E}\langle\uparrow(Y,\mathbb{R}),m\rangle$	$\bot$	$\mathcal{E}\langle\uparrow(Y,X),n\rangle^{\dagger}$

† if $n=m$

$\tau_*(\tau_1,\tau_2)$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{V}\langle X\rangle$	$\mathcal{E}\langle X,n\rangle$
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{V}\langle\uparrow(\mathbb{Z},X)\rangle$	$\mathcal{E}\langle\uparrow(\mathbb{Z},X),n\rangle$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{V}\langle\uparrow(\mathbb{R},X)\rangle$	$\mathcal{E}\langle\uparrow(\mathbb{R},X),n\rangle$
$\mathbb{V}\langle Y\rangle$	$\mathbb{V}\langle\uparrow(Y,\mathbb{Z})\rangle$	$\mathbb{V}\langle\uparrow(Y,\mathbb{R})\rangle$	$\bot$	$\bot$
$\mathcal{E}\langle Y,m\rangle$	$\mathcal{E}\langle\uparrow(Y,\mathbb{Z}),m\rangle$	$\mathcal{E}\langle\uparrow(Y,\mathbb{R}),m\rangle$	$\bot$	$\bot$

$\tau_/(\tau_1,\tau_2)$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{V}\langle X\rangle$	$\mathcal{E}\langle X,n\rangle$
$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{V}\langle\tau_/(I,X)\rangle$	$\mathcal{E}\langle\tau_/(I,X),n\rangle$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{V}\langle\tau_/(R,X)\rangle$	$\mathcal{E}\langle\tau_/(R,X),n\rangle$
$\mathbb{V}\langle Y\rangle$	$\mathbb{V}\langle\tau_/(Y,\mathbb{R})\rangle$	$\mathbb{V}\langle\tau_/(Y,\mathbb{R})\rangle$	$\bot$	$\bot$
$\mathcal{E}\langle Y,m\rangle$	$\mathcal{E}\langle\tau_/(Y,\mathbb{R}),m\rangle$	$\mathcal{E}\langle\tau_/(Y,\mathbb{R}),m\rangle$	$\bot$	$\bot$

$\tau_{a^b}(\tau_1,\tau_2)$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{V}\langle X\rangle$	$\mathcal{E}\langle X,n\rangle$
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{R}$	$\bot$	$\bot$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\bot$	$\bot$
$\mathbb{V}\langle Y\rangle$	$\mathbb{V}\langle\tau_{a^b}(Y,I)\rangle$	$\mathbb{V}\langle\tau_{a^b}(Y,R)\rangle$	$\bot$	$\bot$
$\mathcal{E}\langle Y,m\rangle$	$\mathcal{E}\langle\tau_{a^b}(Y,\mathbb{R}),m\rangle$	$\mathcal{E}\langle\tau_{a^b}(Y,\mathbb{R}),m\rangle$	$\bot$	$\bot$

Fig. 8. Type inference tables for binary operations $\otimes\in\{+,-\}$, $\otimes=*$, $\otimes=/$, and exponentiation a^b.

on the operation ($\ominus$ or $\otimes$) and the types of the operand(s). This also simplifies the implementation of the type system and makes it extensible. The type inference for unary operators $\ominus\in\{-,\sqrt{\ },!\}$ can be summarized as follows: The logical not ! can be applied only to boolean arguments $e:\mathbb{B}$ and its result is boolean as well. The unary minus is applicable for numerical expressions with $\tau\in\{\mathbb{Z},\mathbb{R}\}$ and will not change their types. Similarly, the square root operator can be applied to arguments of numerical type and the result will be a real number (or a runtime exception if the result would be a complex number, which is not included in the current static type system). The more detailed type-inference tables for binary arithmetic expressions $\tau_\otimes$ can be found in Figure 8. In the tables, abbreviated forms for the types are used, where $\mathcal{E}$ denotes a particle property or discretized data from a field. The integers n are m denote the size of the data (i.e., the number of elements in the vector). Additionally, $\uparrow(\tau_1,\tau_2)$ denotes the least common super-type of τ_1 and τ_2. Note that, for the sake of brevity, we did not include the inference tables of the remaining expressions. If $\tau_\otimes$ is undefined, then it is denoted by $\bot$. Moreover, the rules that end on Acc (*Acc) define the type inference for scalar and vector access to particle properties, which is a core task of the system.

Type Errors. All expressions matching one of the rules presented above are *well-formed*, and their type can be inferred by the system. However, it may occur that the user enters a faulty expression for which no type inference is possible, yielding a *typing error*. In this case, PPME has to communicate the error to the developer and provide meaningful information on where the error is located. To formalize this, we introduce an *error type* $\mathbb{E}$ used as a result for non-well-formed expressions (Plotkin 1981, 2004). There are different causes for typing errors, e.g., incompatible types or undefined behavior. For instance, the exponentiation of a scalar with a string is not a meaningful mathematical operation and should yield a typing error in the corresponding expression. Error detection is not limited to arithmetic expressions, but covers domain-specific concepts as well. Furthermore, errors might be propagated, invalidating the parenting expression. To support this in the type system, we add two extra rules ErrUnary and ErrBin. A typing error $\mathbb{E}$ occurs when type resolution fails, i.e., if $\tau_\ominus(\tau')$ and $\tau_\otimes(\tau_1,\tau_2)$ are undefined.

4.3 Dimension Annotations

Adding the notion of measurement units to a programming language benefits software developers in many ways, especially in checking the physical consistency of equations. Verifying dimensional integrity prevents errors in expressions that may be hard to detect otherwise. The language and type system extension for dimension annotations hence adds an additional level of analysis to detect inconsistencies at compile-time. Early work by Karr and Loveman (1978) presented a "units calculus," a method to manage relationships and conversions of units, to be incorporated in programming languages. In Cmelik and Gehani (1988) and Umrigar (1994), the authors extended the idea of measurement units to general *dimensional analysis*, covering dimensional classes of units, e.g., length or mass, meaning that quantities with the same dimension but different units differ only by a conversion factor. Dimensional analysis fits neatly into the concept of type inference in functional languages, establishing the base for units and dimensions in functional languages (Wand and O'Keefe 1991; Kennedy 1994, 1997; Hayes and Mahony 1995).

In PPME, we consider dimensions and units as additional annotations to types and expressions that are processed by an extended type system, with $\mathcal{I}$ the set of dimensions supported by this system. Dimensions without specification, such as length l, mass m, or time t, are called a *fundamental*. We denote fundamental dimensions with $\check{\delta}$ and the set of fundamental dimensions as $\check{\mathcal{I}} \subseteq \mathcal{I}$. Additional *derived dimensions* δ, such as acceleration or force, can be composed from others by means of multiplication and exponentiation, e.g., for acceleration $a = l \cdot t^{-2}$. While all derived dimensions can be composed from fundamental dimensions only, PPME also allows definitions from other derived dimensions to simplify notation. This is described by the following grammar:

$$\delta ::= \check{\delta} \mid \delta_1 \cdot \delta_2 \| \ \delta^n \,,$$

where $\hat{\delta} \in \hat{\mathcal{I}}$, $n \in \mathbb{Z}$, and derived dimensions δ_1, δ_2, and δ. To make dimensions comparable, they are represented in *base form*, i.e., as a combination of $k > 0$ fundamental dimensions $\check{\delta}_i$ raised to some integer exponent n_i where each $\check{\delta}_i$ occurs at most once. A base form of δ can be constructed by recursively replacing all derived dimensions in δ by their declaration, and grouping all occurrences of the same fundamental dimension $\check{\delta}_i$ in a single equivalent power representation $\check{\delta}_i^{n_i}$. We denote the expansion and base form of δ by $\lceil \delta \rceil$ with

$$\lceil \delta \rceil := \{\check{\delta}_1^{n_1}, \check{\delta}_2^{n_2}, \ldots, \check{\delta}_k^{n_k}\} \,.$$

Based on this definition, two dimensions δ_1 and δ_2 *match* if $\lceil \delta_1 \rceil = \lceil \delta_2 \rceil$, denoted by $\delta_1 \equiv \delta_2$.

Dimensions can be easily integrated into the PPME type system by extending it with dimension-specific rules and retaining the original inference mechanism. Given a type τ and a dimension δ, we denote the *annotated type* by $\hat{\tau} = [\tau; \delta]$. In particular, any type τ can be annotated with the *empty dimension* $\emptyset$ without changing semantics by using $e : [\tau; \emptyset]$ instead of $e : \tau$. The annotation of metadata to types or literals is denoted by curly braces, i.e., $\tau\{\delta\} : [\tau; \delta]$, and $e\{\delta\} : [\tau; \delta]$ instead of $e : \tau$. Moreover, for dimension inference, we use a notation similar to the type inference table in Figure 8. For instance, $\delta = \mathcal{I}_\otimes(\delta_1, \delta_2)$ denotes that δ is inferred from the operand dimensions δ_1 and δ_2 and the operation $\otimes$.

Finally, the original type rules shown in Figure 7 need to be adapted. As this adaptation is mostly straightforward, we only show the most relevant rules in Figure 9. The rules for handling variable references (Var) and variable declarations (VarDecl and VarInit) have been expanded with annotated dimensions. Assignment expressions (Assign) now take the annotated dimension into account. The general scheme for unary (Unary) and binary operations (BinOp) is, likewise, extended for annotated types. Besides type inference ($\tau_\ominus$ and $\tau_\otimes$), dimensions are inferred through $\mathcal{I}_\ominus$ and $\mathcal{I}_\otimes$. The rule ErrDim exemplary shows the additional potential for error detection introduced by dimensions: even if types match, a dimension error is still detectable.

$$\textsc{Var}\ \frac{\Gamma(v) = [\tau;\delta]}{\Gamma \vdash v : [\tau;\delta]} \qquad \textsc{VarDecl}\ \frac{}{\Gamma \vdash \tau\{\delta\}\, x : \Gamma \cup \{x = [\tau;\delta]\}} \qquad \textsc{VarInit}\ \frac{\Gamma \vdash e : [\tau';\delta'] \quad \tau' \leq \tau \quad \delta' \equiv \delta}{\Gamma \vdash \tau\{\delta\}\, x = e : \Gamma \cup \{x = [\tau;\delta]\}}$$

$$\textsc{Assign}\ \frac{\Gamma \vdash x : [\tau;\delta] \quad \Gamma \vdash e : [\tau';\delta'] \quad \tau' \leq \tau \quad \delta' \equiv \delta}{\Gamma \vdash x = e : [\tau;\delta]} \qquad \textsc{Unary}\ \frac{\Gamma \vdash e : [\tau;\delta] \quad \tau_{\ominus}(\tau) \neq \bot \quad \mathcal{I}_{\ominus}(\delta) \neq \bot}{\Gamma \vdash \ominus e : [\tau_{\ominus}(\tau); \mathcal{I}_{\ominus}(\delta)]}$$

$$\textsc{BinOp}\ \frac{\Gamma \vdash e_1 : [\tau_1;\delta_1] \quad \Gamma \vdash e_2 : [\tau_2;\delta_2] \quad \tau_{\otimes}(\tau_1,\tau_2) \neq \bot \quad \mathcal{I}_{\otimes}(\delta_1,\delta_2) \neq \bot}{\Gamma \vdash e_1 \otimes e_2 : [\tau_{\otimes}(\tau_1,\tau_2); \mathcal{I}_{\otimes}(\delta_1,\delta_2)]}$$

$$\textsc{ErrDim}\ \frac{\Gamma \vdash e_1 : [\tau_1;\delta_1] \quad \Gamma \vdash e_2 : [\tau_2;\delta_2] \quad \tau_{\otimes}(\tau_1,\tau_2) \neq \bot \quad \mathcal{I}_{\otimes}(\delta_1,\delta_2) = \bot}{\Gamma \vdash e_1 \otimes e_2 : \mathbb{E}}$$

Fig. 9. Type rules for dimension-annotated expressions in PPME.

$$\dfrac{\Gamma \vdash p{\rightarrow}v : [\mathbb{R}, v] \qquad \dfrac{\Gamma \vdash 0.5 : [\mathbb{R}, \emptyset] \qquad \dfrac{\dfrac{\Gamma \vdash p{\rightarrow}a : [\mathbb{R}, a] \qquad \dfrac{\Gamma \vdash p{\rightarrow}F : [\mathbb{R}, m \cdot a] \quad \Gamma \vdash mass : [\mathbb{R}, m]}{\Gamma \vdash p{\rightarrow}F/mass : [\mathbb{R}, a]}\,(1)}{\Gamma \vdash p{\rightarrow}a + p{\rightarrow}F/mass : [\mathbb{R}, a]}\,(2)}{\Gamma \vdash 0.5 * (p{\rightarrow}a + p{\rightarrow}F/mass) : [\mathbb{R}, a]}\,(3) \qquad \Gamma \vdash delta\_t^2 : [\mathbb{R}, t^2]}{\Gamma \vdash 0.5 * (p{\rightarrow}a + p{\rightarrow}F/mass) * delta\_t^2 : [\mathbb{R}, a \cdot t^2]}\,(4)}{\Gamma \vdash p{\rightarrow}v + 0.5 * (p{\rightarrow}a + p{\rightarrow}F/mass) * delta\_t^2 : \mathbb{E}}\,(E)$$

$$(1)\ \tau_{/}(\mathbb{R},\mathbb{R}) = \mathbb{R}, \mathcal{I}_{/}(m \cdot a, m) = a \qquad (2)\ \tau_{+}(\mathbb{R},\mathbb{R}) = \mathbb{R}, \mathcal{I}_{+}(a,a) = a \qquad (3)\ \tau_{*}(\mathbb{R},\mathbb{R}) = \mathbb{R}, \mathcal{I}_{*}(\emptyset,a) = a$$
$$(4)\ \tau_{*}(\mathbb{R},\mathbb{R}) = \mathbb{R}, \mathcal{I}_{*}(a,t^2) = a \cdot t^2 \qquad (E)\ \tau_{+}(\mathbb{R},\mathbb{R}) = \mathbb{R}, \mathcal{I}_{+}(v, a \cdot t^2) = \bot$$

Fig. 10. Example deduction using the extended type system. The applied type inference rule $\tau_{\otimes}$ and dimension inference rule $\mathcal{I}_{\otimes}$ for respective steps (1) to (4), and (E) are shown in the box. The failing dimension inference rule is marked in red.

As an example for applying the type system, consider the following PPME expression that has been modified from the Lennard-Jones case study discussed in Section 5.3:

$$p{\rightarrow}v + 0.5 * (p{\rightarrow}a + p{\rightarrow}F/mass) * delta\_t^2\,.$$

Here, p refers to a particle whose properties v (velocity) and a (acceleration), and the force field F are used together with the free variables *mass* and *delta_t* (time) to compute an update in a larger simulation. We introduced a small error into the expression: *delta_t* has an exponent of 2 instead of 1. Since this does not have an impact on the overall type of the expression ($\mathbb{R}$), a conventional type system cannot detect this error. However, using dimensions, the problem becomes discoverable, as shown by the deduction depicted in Figure 10. In step (1), the type and dimension of the subexpression $p{\rightarrow}F/mass$ is deduced from its compartments ($[\mathbb{R}, a]$). Step (2) deduces the type of the enclosing addition of $p{\rightarrow}a$ and $p{\rightarrow}F/mass$ (again $[\mathbb{R}, a]$). In step (3), the type remains the same because of multiplication with the constant 0.5. Step (4) computes the type of the multiplication with $delta\_t^2$ as $[\mathbb{R}, a \cdot t^2]$. Finally, in step (E), the type system discovers that the outermost sum is infeasible, since $[\mathbb{R}, v]$ and $[\mathbb{R}, a \cdot t^2]$ are incompatible, deducing error type $\mathbb{E}$. In contrast, if the expression would have been correct, the calculus would have derived $[\mathbb{R}, v]$ as the overall type.

5 THE PPM ENVIRONMENT: ARCHITECTURE AND IMPLEMENTATION

The *productivity* of scientific programmers can be increased by providing high-level abstractions for computational models, such that the developer is not bothered with details of the programming language or the underlying hardware architecture. While *quality* is hard to measure, an IDE can check for common errors up-front and present the developer with meaningful warnings and error messages. Additionally, static program analysis, paired with domain knowledge, can be used to improve performance, accuracy, and/or efficiency of simulations. Incorporating third-party

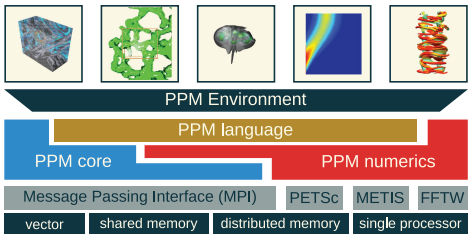


Fig. 11. PPME: New access layer to the underlying PPML.

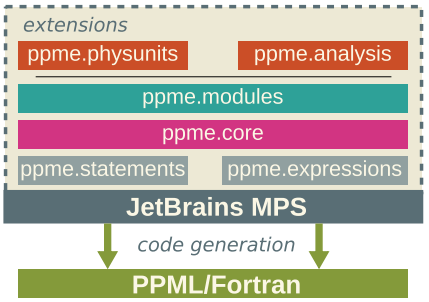


Fig. 12. Modular architecture of PPME.

applications allows us to reuse established tools for analysis and program transformation instead of reimplementing their features. Full access to the underlying language and implementation enables advanced scientific programmers to leverage their knowledge to add new language-level features and to have full control over program performance. All of these features are available in PPME.

5.1 Internal Structure of PPME

Based on the Meta Programming System (MPS), PPME adds an additional layer on top of the existing PPML stack (Sbalzarini et al. 2006; Awile et al. 2013) and does not require any adaptation in the underlying framework. It generates source code against PPML, and therefore makes use of the established workflow, using PPML as an intermediate representation. Figure 11 illustrates how PPME fits between the user program and the PPM middleware. Application developers interact with the development environment to implement particle methods and the related configuration files. PPML's original purpose of hiding technical details, specific realizations, and the explicit target platform is preserved. However, PPME offers a more consistent DSL syntax and incorporates domain-specific elements as first-class concepts.

In PPME, the domain metamodel of Section 4.1 is organized in language packages, called *solutions* in MPS. The clean separation between different sub-languages enables a good separation of concerns. The lower layers form a base DSL with general language constructs such as expressions, literals, and statements. These concepts are reused in the upper layers to define domain-specific language concepts for particle-mesh methods on top of the base language. Reusing lower layers and their extension is key in the design of PPME, enabling easier maintenance and custom extensions for specific cases as plug-ins.

Figure 12 provides a schematic view of PPME's internal language stack. The bottom layers form the main DSL while sub-languages are used to keep the implementation modular and maintainable. At the interface to the underlying PPM library, MPS manages code transformation and generation. The top layer is open to new application-specific extensions, e.g., for particle-based image processing (Afshar and Sbalzarini 2016). The languages packages of PPME (cf. Figure 12) are:

ppme.expressions. This package provides general expressions as can be found in most programming languages, e.g., mathematical and logical expressions, and literals for integer and floating-point numbers. Moreover, the base types available in PPME are defined in this package, as well as essential parts of the type system introduced in Section 4.2. As already mentioned previously, the main purpose of the static type system is to detect illegal expressions early, at compile time or while editing. The PPME editor therefore instantaneously analyzes the program using the

type-inference rules. When an error type $\mathbb{E}$ is derived, the editor displays an error mark and a cause-related error message, if the cursor is hovered over the erroneous expression.

ppme.statements. The statements sub-language contains a basic set of imperatives, such as expression statements, if-else clauses, and loops. Furthermore, variable declarations and references are part of this package. The type system is enriched with variable support where necessary (cf. Section 4.2.3). Overall, the elements of this language are universal, since they are independent of the domain they are used in.

ppme.core. The core package contains most elements specific to particle methods. It extends the solutions for expressions and statements by adding new domain-specific types, expressions, and statements. Selected constructs of PPML are reflected in PPME while remaining consistent with the base language's concepts. For instance, the *timeloop* construct of PPML is available in this package.

ppme.modules. A module in PPME is the top-level structure for client programs written in PPME. It contains the simulation code and optional specifications for imported control parameters. A module translates to a PPML client, but the IDE can use additional knowledge about the domain better than PPML, e.g., by referencing external control files and inspecting the code.

ppme.lang. This package is an MPS *devkit* that contains the above base languages of PPME (not shown explicitly in Figure 12). In MPS, devkits group interconnected languages as one unit. Hence, to get the base functionality of PPME's language it suffices to include the devkit in an MPS project, covering all language dependencies.

In addition to these base languages, PPME provides optional solutions that add dimensions and physical units into program specifications, and for the integration of external analysis tools. Both serve as examples for further extensions tailored to specific use-cases:

ppme.physunits. The optional physical-units integration enables developers to annotate further meta information to variables and constants. This includes means for adding dimensions and physical unit specifications as an additional extensible layer in the type system (cf. Section 4.3).

ppme.analysis. The analysis language consists of an exemplary binding of *Herbie* as an external analysis tool for improving floating-point expressions (Panchekha et al. 2015). We elaborate a general framework enabling the access of custom tools in the environment. More details on this tool integration will be given later in Section 6.

5.2 Code Generation

The code-generation process in PPME involves an integrated transformation and analysis chain. This process is sketched subsequently, followed by two concrete example transformation steps.

5.2.1 Transformation Process. Code generation in PPME is implemented via several model-to-model transformations refining the program, and a final text-generation stage that produces source code in the PPML target language. Models in MPS are directed graphs with type annotations derived from the metamodel. The graphs have a distinct spanning tree, which in general corresponds to an abstract syntax tree. Model-to-model transformations map an input graph to an output graph, where the output graph may use the same or different type annotations given by the same or another metamodel. Models must adhere the structural constraints defined by their metamodels. An excerpt of the PPME metamodel is given in Figure 4. During the transformation process, the internal graph-based representation of the program is enriched with additional information that is explicitly required to generate the output in the target language, e.g., a list of variables accessed in a loop can be derived from the loop's body and made available explicitly for further processing. In general, the concept of staged language processing is advantageous, for example, to yield different output representations of an input program, or for transformation chaining.

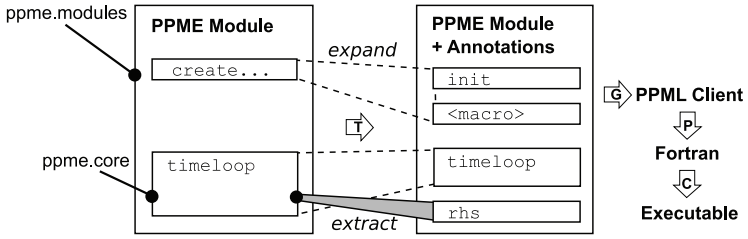


Fig. 13. Domain-specific abstractions in a PPME module are transformed (**T**) into lower-level representations. From this enriched module, PPML client code is generated (**G**), which is subsequently processed (**P**) to Fortran code and compiled (**C**) to a binary linking against the PPM Library.

Dependencies between transformation steps are resolved automatically so that a global transformation sequence for a given program can be computed. This allows adding new features and further transformations to extend PPME without affecting other components.

To avoid unnecessary overhead during the generation phase, textual output is produced in only two cases: (a) when the final output in the target language is generated, and (b) when external tools and analyses require textual input. This restriction enables full control of the transformation phase within MPS, taking into account the enrichments of various transformations and results of external components, such as Herbie and other tools. The produced source code can then be compiled using a regular compiler.

The code generation process is illustrated in Figure 13. It starts with a simulation program implemented using the domain metamodel introduced in Section 4.1. The module contains domain-specific concepts, such as a `timeloop` and various constructs to define particle-based simulations. In Figure 13, the `timeloop` statement is analyzed in the first transformation stage and a PPML right-hand side specification (cf. Figure 2, Lines 53–65) is extracted. Similarly, the creation of particles is expanded to several initialization and macro calls in the representation of the module, which is closer to the target language.

The majority of the model-transformations are part of the top-level package `ppme.modules`. Various mapping scripts are used to *pre-process* the input-model for collecting information required in later transformation steps. To produce the intermediate code (i.e., PPML code), MPS' *text gen* capabilities are used (cf. MPS - 3.2 - Documentation (2015a)). For each language concept, a text generation component can be specified, defining its textual representation, e.g., printing the name of a variable (`VariableReference`), or emitting the code for a loop statement.

We use several transformation scripts to prepare the text-generation phase. Therefore, we have defined multiple intermediate models resembling the macros and first-class language constructs in PPML to stepwise refine the input model. This includes collecting information about the differential operators used in equations, adding explicit discretization statements for them, creating and populating right-hand side declarations of PPML, adding ODE declarations, managing control files, expanding random-number initialization, deriving field and property declarations, transforming foreach loops into their PPML counterparts, and adding PPML-specific type annotations. Exemplarily, we discuss the construction of right-hand-side declarations.

5.2.2 Example Transformation. The `populateRHS` transformation is responsible for extracting right-hand-side definitions from `deqn` statements, which model differential equations. In PPME, these equations can be written directly in code. To transform them into PPML, we use `RHSMacros`, which represent the right-hand side definitions in the target language and, in turn, can be transformed into PPML macro code. The `deqn` statements are matched and extracted by the code

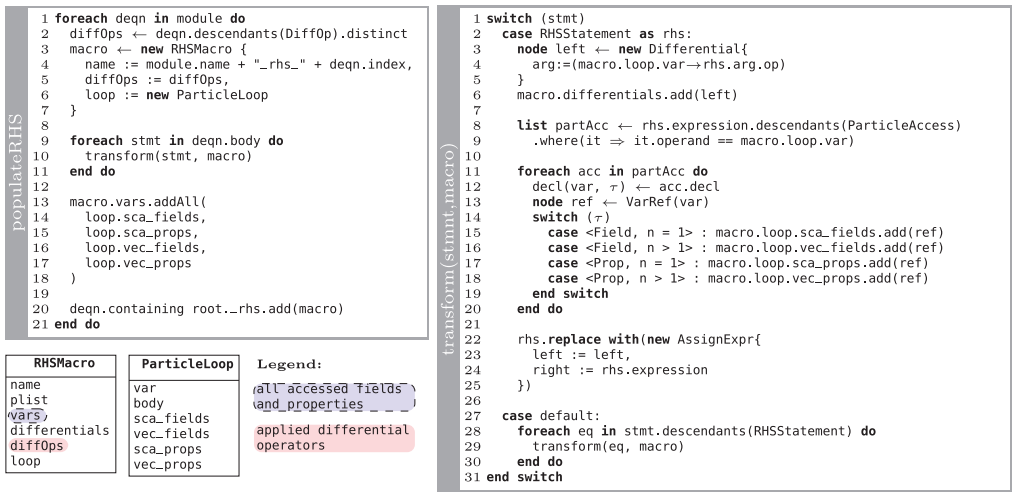


Fig. 14. Excerpts from the script that composes a PPML RHSMacro from a PPME deqn specification.

generator using transformation scripts in MPS' built-in scripting language. The scripting language is statically typed and borrows ideas from object-oriented and functional programming that are well-suited for model transformations, including higher-order functions, and type-based selectors on trees and lists. For example, the descendants selector visits a tree or list and collects references to nodes of a given type, like iterators in an object-oriented language. In addition, higher-order functions such as map or where allow developers to map or filter lists by applying a given (anonymous) function, like in a functional program. The left panel of Figure 14 shows the script that implements the populateRHS transformation. It iterates over all blocks of deqn statements in a program, and derives corresponding RHSMacros. A key issue in this process is to identify the used differential operators contained in a deqn statement (l. 2) and the accessed variables (ll. 13–18). A PPML particle-loop, evaluating the differential operators over these variables, is assembled by transforming each statement in the deqn via delegation to transform (ll. 9–11). Finally, the created macro object is added to the model root and then used for generating the PPML right-hand side.

The script excerpt in the right panel of Figure 14 shows the body of the recursive transform() function. It takes a statement of a differential equation and enriches the given macro with information. For each differential equation statement $\frac{\partial c \to f}{\partial t} = e$ (stmt), the affected particle attribute (f) is extracted (ll. 2–6). Subsequently, the accessed particle fields and properties are extracted and explicitly added to the RHSMacro (ll. 11–20). Finally, the differential equation is translated to an assignment expression that replaces the original definition in the model and is later translated to PPML statements for the generated code of the right-hand side.

Figure 15 shows a comparison of the original PPME block and the resulting PPML code for the Gray-Scott example (cf. Section 3). It integrates the governing equations discretized over the particle list c using the fourth-order Runge-Kutta method ("rk4") for time integration. In PPME, a developer conveniently defines equations over attributes of a particle list c. The IDE automatically extracts the required information. First, two applications of differential operators, $\nabla^2 c \to U$ and $\nabla^2 c \to V$, are identified. The local variables dU and dV are inserted to hold the intermediate result of applying the operators. Both the particle loop and the right-hand-side block itself hold derived information about the accessed particle fields, U and V. Furthermore, dU and dV are treated like other scalar fields. Note that the access of particle list attributes (c→U) is transformed to access of

PPME

```
deqn method "rk4" on c
  ∂c→U/∂t = constDu * ∇² c→U - c→U * c→V² + F * (1.0 - c→U)
  ∂c→V/∂t = constDv * ∇² c→V + c→U * c→V² + (F + kRate) * c→V
end deqn
```

PPML

```
rhs grayscott_rhs_0(U=>c, V)
  get_fields(dU, dV)

  dU = apply_op(L, U)
  dV = apply_op(L, V)

  foreach p in particles(c) with positions(x) sca_fields(U, V, dU, dV)
    dU_p = constDu * dU_p - U_p * (V_p**2) + F * (1.0 - U_p)
    dV_p = constDv * dV_p + U_p * (V_p**2) - (F + kRate) * V_p
  end foreach
end rhs
```

Fig. 15. PDE specification in PPME (top) and the generated right-hand side in PPML (bottom).

particle attributes by inserting a loop over particles and accessing attributes of the loop variable `U_p`. The example demonstrates some of the key benefits of our approach over the original PPML code: since all required information is extracted by the PPME compiler, redundant statements such as `get_fields`, `apply_op` and `sca_fields` are avoided, which leads to less code, less compile-time errors, and an improved readability. Readability is further improved by the PPME editor natively supporting basic mathematical notation, such as the Nabla operator ∇ and the partial derivative ∂.

5.3 Case Studies

To demonstrate the capabilities of PPME, we use the same two simulations as case studies that were already considered for PPML (Awile et al. 2013). The first one, the Gray-Scott reaction-diffusion system as presented in Section 3.1, is an example of a simulation of a continuous deterministic model. The second one, Lennard-Jones molecular dynamics is an example of a simulation of a discrete deterministic model. An N-body simulation as a third example further illustrates the initialization of particles from external data.

5.3.1 Gray-Scott Reaction-Diffusion System. The PPME program for the Gray-Scott simulation is shown in Figure 16. It follows the typical structure of a particle-based simulation, starting with the initialization of topology and particles, followed by the simulation loop. The notation in PPME is concise and close to the domain idiom. The program starts with the module definition and the referenced runtime constants. At the beginning of the simulation, topology, particles, and neighbor lists are set up. The time steps are contained in the `timeloop` and are solely defined through the differential equations to be solved. For the equation block, the developer has to specify the particle list the equations are working on, and the time-stepping method. Note that the continuous fields U and V are automatically discretized on the particle list during code generation.

5.3.2 Lennard-Jones Molecular Dynamics. Lennard-Jones is an instance of molecular dynamics (Frenkel and Smit 2001), an item-based simulation to study molecular processes. The atoms are directly represented as particles, located in continuous space. Pairwise potentials between atoms define the continuous forces acting on them. While the basic algorithm for the simulation, i.e., computing pairwise interactions of particles and updating their positions and properties, remains the same, the exact definition of the forces is specific to the application. A classical force definition is given by the Lennard-Jones potential, which is suitable for describing inert gases. The pairwise force between atoms depends on the distance between them (r), the depth of the potential well (ε), and the fall-off distance (σ) of the interaction potential. Particle properties such as acceleration (a) or velocity (v) change according to the forces, causing the particles to move. Additionally, a cutoff radius to ignore negligibly small long-range interactions is applied.

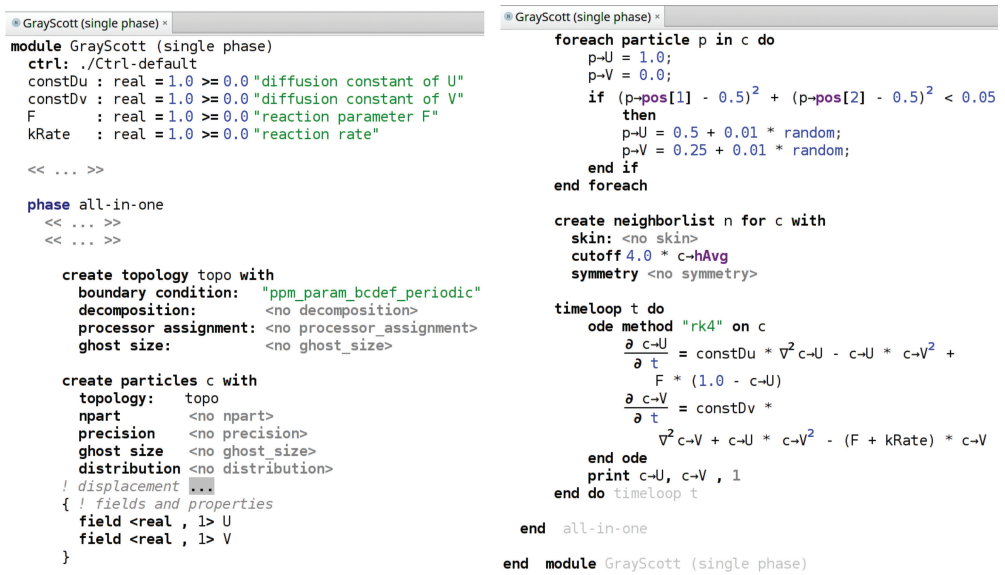


Fig. 16. Gray-Scott simulation program in PPME.

The essential part of simulating the potential is located in the `timeloop` depicted in Figure 17. Therein, the force acting on the particles due to pairwise interactions is computed and applied. The loop can be divided into four sections (①–④). First, the particle positions ($p{\rightarrow}pos$) are updated based on the values of velocity ($p{\rightarrow}v$) and acceleration ($p{\rightarrow}a$) (cf. ①). After the particle positions change, the boundary condition must be imposed, followed by updating the mappings and neighbor list (cf. ②). The block of two nested particle loops implements the actual particle–particle interactions (cf. ③). For each particle p the pairwise interaction with all nearby particles q, retrieved via `neighbors(p, nlist)`, is computed. The force $F = -\nabla E$ acting between two particles and the potential (or energy) E are given by

$$\vec{F}(r) = 24\varepsilon r\left(2\frac{\sigma^{12}}{r^7} - \frac{\sigma^6}{r^4}\right), \qquad E(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]. \tag{4}$$

This corresponds to the lines with assignments to `dF` and $p{\rightarrow}E$, respectively, where `r_s_pq2` corresponds to the squared distance r^2 between p and q. The last update on the particle list modifies the velocity as a consequence of the new force (cf. ④).

In this example, we also use dimensions to further improve static error detection. A *dimension declaration* in PPME resides in a special file owned by a model, where each declaration contains an identifier <d>, an optional specification <spec>, and an optional suggestive name <desc>. Figure 19 shows the declaration of fundamental and derived dimensions. From the three fundamental dimensions length (l), time (t), and mass (m), convenient notations for velocity (v) and acceleration (a) are derived. Note that this specification is not bound to this example and can be reused in any other PPME project.

Figure 18 shows the annotated particle properties, which are used by PPME's type system to derive expression types. The type system corresponds to the formalization introduced in the previous section. It enables capturing type and dimension errors right in the editor. Errors are reported to the user where they occur, as shown in Figure 20 using the deduction in Figure 10. The outer

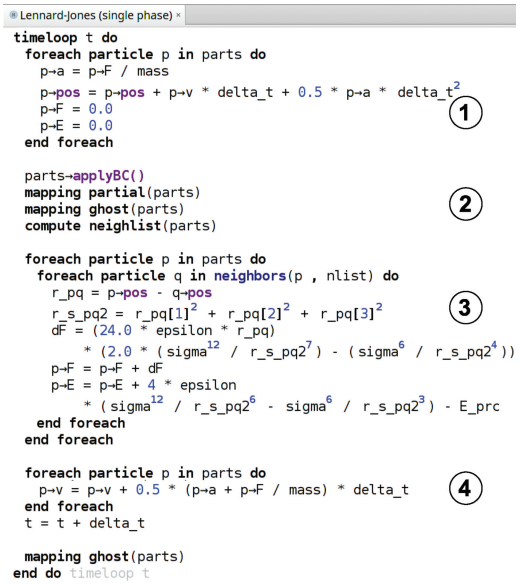


Fig. 17. The simulation loop body for the Lennard-Jones dynamics.

```
{ ! fields and properties
  property <real{v} , ppm_dim , "velocity" , true> v
  property <real{a} , ppm_dim , "acceleration" , true> a
  property <real{m·a} , ppm_dim , "force" , true> F
  property <real{l·m·a} , 1 , "energy" , true> E
  property <real{l} , 3 , "position" , <no zero>> pos
}
```

Fig. 18. Dimensions annotated to particle properties in the Lennard-Jones example.

```
units :
  l := <no spec> ( length )
  t := <no spec> ( time )
  m := <no spec> ( mass )
  v := l·t^-1( velocity )
  a := l·t^-2( acceleration )
```

Fig. 19. Declarations of base dimensions (length, mass, and time) with velocity and acceleration as derived dimensions.

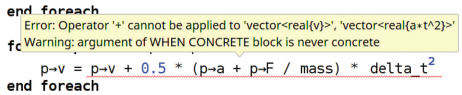


Fig. 20. User notification of an error caused by incompatible dimensions.

addition is highlighted, and the information states that the operation cannot be applied to operands with given (annotated) types $[\mathbb{R}, v]$ and $[\mathbb{R}, a \cdot t^2]$.

5.3.3 N-body Simulation. As a third case study, we implement an N-body simulation of two galaxies using particle methods. As the model structure of this example is similar to the Lennard-Jones example above, we skip the corresponding details of the code and focus on another important aspect of PPME: its interface with the underlying Fortran language.

PPME is designed as a standalone DSL, nevertheless unanticipated use cases can be supported by inline code statements. In the N-body simulation, the initial particle data need to be loaded from an external data source `data.tab`. As PPME has no built-in functionality to import data from this type of file, it has to be specified using custom code. This can be achieved through an `InlineCodeStatement`, which supports inlining arbitrary Fortran or PPML code directly into the program. Figure 21 shows how `data.tab` is read and into PPME's data structures. The PPML code is located within a pair of squared brackets, which demarcate it from the rest of the program. The code has direct access to the matrix `parts_data` as well as to the declared fields `v` and `m`. During this custom initialization, the data are first loaded into the matrix (which corresponds to a Fortran array) and afterwards copied to the corresponding fields.

Notice that the inlined PPML code is not analyzed by PPME. Errors may therefore be introduced by the developers that propagate to later stages in the compile chain. However, such code can be conceptualized easily by extending the language and converting it into an MPS generator, which is one of the central ideas in language-oriented programming.

6 NUMERICAL OPTIMIZATIONS AND TOOL INTEGRATION

Applications in science and engineering often depend on floating-point arithmetic in calculations to approximate real arithmetic. In this section, we therefore introduce an accuracy optimization

```
nbody ×
  create particles pl with
    topology:    topo
    npart        <no npart>
    precision    <no precision>
    ghost size   ghost_size
    distribution <no distribution>
  { !displacement
    << ... >>
  }
  { ! fields and properties
    property <real , 3 , "velocity" , <no prec> , true> v
    property <real , 1 , "mass" , <no prec> , true> m
  }

  matrix < real > parts_data

  ! - initialize particles ----------------------------------------------
    allocate(parts_data(pl%Npart, 7))
    open(10, file='data.tab', action='read')
    do i=1,pl%Npart
      read(10, *) parts_data(i,:)
    end do
    close(10)

    print *, "== Initializing particle set ============================="

    foreach p in particles(pl)
         with positions(x, writex=true) vec_props(v,a) sca_props(m)
      m_p = parts_data(p,1)
      x_p(:) = parts_data(p,2:4)
      v_p(:) = parts_data(p,5:7)
      a_p(:) = 0
    end foreach

  ! - end initialize particles ------------------------------------------
```

Fig. 21. Particle initialization using PPML inline code.

for floating-point expressions that is integrated into PPME. This serves as an example of how to extend PPME by existing tools, thereby demonstrating the possibilities offered by the high-level expressions of the language. In a similar way, other common optimizations can be added in the future, for example, loop transformations to improve data locality and performance (Lam and Wolf 1991; Luporini et al. 2015). Loop optimizations are orthogonal to the floating-point transformation described in this section. We expect loop unrolling, vectorization, and other optimizations to be enabled via annotations or pragmas, aided by the high-level information about dependencies available at the level of the DSL.

In the case of floating-point computations, a compiler can trade *accuracy* for *performance*. Such optimizations often rely on *abstract interpretation* for preserving semantics. The abstract semantic can be used to build program equivalent graphs, inspect them with regard to the desired optimization target, and enable efficient detection of appropriate rewrites (Ioualalen and Martel 2012). As an example of such optimizations, we adopt *Herbie*[1] as a recent approach to automatically improve floating point expressions (Panchekha et al. 2015). Herbie relies on heuristics to estimate and localize rounding errors at sample points. Once low accuracy (e.g., numerical extinction) is detected, Herbie attempts to improve the program by rewriting inaccurate expressions using a rule database. Thereafter, if possible without loss of accuracy, the expressions are simplified. Finally, Herbie may apply series expansions for inputs around zero or near infinity to better approximate the result. This process is repeated iteratively so that new candidate expressions are yielded after each iteration, keeping only the programs that achieve the best accuracy at least at one sample point.

[1] https://github.com/uwplse/herbie.

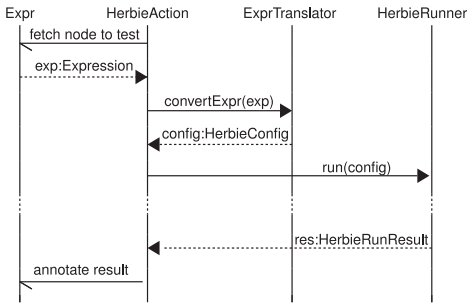


Fig. 22. Sequence diagram for the execution of a Herbie analysis for a single expression.

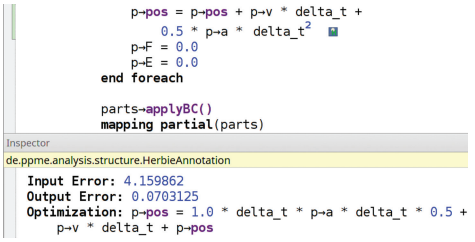


Fig. 23. Inspection view of an annotated expression with the results reported by Herbie.

Finally, Herbie uses one or more candidates to achieve an improvement of accuracy over all sample points.

6.1 Tool Integration

We integrated Herbie as a plugin into PPME. MPS provides convenient configuration languages to define *plugin solutions* and to specify their behavior. The solution of our Herbie plugin comprises a preference page, an action object that executes Herbie in a separate process, and a *mapping script* that collects references to expression nodes in the PPME program. Since the algorithms for accuracy optimization are computationally expensive, expressions are not inspected automatically but must be flagged for evaluation by the user. Users can trigger the analysis and transformation process for the active PPME editor, and the result is then annotated to the corresponding fragments in the code.

Expression annotation works via the MPS intentions dialog (Alt + Enter). A small icon indicates that an expression is marked for analysis (cf. first line in Figure 23). All marked expressions are transformed into a prefix notation that matches Herbie's input language *Racket*, built-in operations and data types. PPME maintains the translated expression as a string, a reference to the original expression node and its identifier, as well as a table of variables. For the expression highlighted in Figure 23 the following test case is generated:

```
(herbie-test (p_pos p_a delta_t p_v) "2430378650379961582"
  (+ (+ p_pos (* p_v delta_t)) (* (* 0.5 p_a) (expt delta_t 2))))
```

where the `herbie-test` macro is called with a list of variables, a unique name, and the actual translated expression.

Figure 22 illustrates the whole process (steered by `HerbieAction`) responsible for fetching the nodes to test, analyze the node, and write the result back as a UML sequence diagram. The result of a Herbie execution is summarized and displayed to the user in the inspection view as shown in Figure 23. The original expression is annotated with additional information, i.e., the error of the input and output expressions, and the optimized computation regime for the arithmetic expression. This allows inspecting the result generated by Herbie, and it keeps the original expression in place without modifying it. Instead, PPME users are responsible for replacing the annotated node by the optimized one before the text generation phase. This prevents unwanted expression rewriting. Note that the generic design of the execution model and the result container allow reusing the same setup for other tools by extending the existing framework. Moreover, analysis and code generation remain separate processes.

Input Error	Output Error
7.1231523	0.08203125

$$\frac{\partial c \to U}{\partial t} = D_u \cdot \nabla^2 c \to U - c \to U \cdot c \to V^2 + F \cdot (1.0 - c \to U)$$

```
<    dU_p = D_u*dU_p - U_p*(V_p**2) + F*(1.0_mk-U_p)
---
>    dU_p = (((constDu * dU_p) - ((U_p * V_p) * V_p)) + ((1.0_mk - U_p) * F))
```

Fig. 24. Exemplary improvements for an expression taken from the Gray-Scott example.

6.2 Accuracy Optimization

We investigate the improvements in accuracy for two of the case-studies presented in Section 5.3, the Lennard-Jones simulation (LJ) and the Gray-Scott reaction-diffusion system (GS). We annotate several expressions in each program and execute them with and without optimization. Figure 24 shows the analysis result for $\frac{\partial U}{\partial t}$ for the GS example, including input and output error, the original expression, and the difference in the generated source code. The input expression has an average error of seven bits (cf. Panchekha et al. (2015)), and Herbie was able to nearly remove inaccuracies by expanding and redistributing terms. We compare the numerical results for simulations with $t_{start} = 0$, $t_{end} = 4{,}000$, and $\Delta t = 0.5$. The computed values for U and V differ in the last 4–7 of 17 significant digits, which confirms that the changes have an impact. In our visualization (cf. Figure 3), the differences are not noticeable. However, a longer simulation time may yield visible differences.

For the LJ case study, we investigate several expressions, this time with and without considering known restrictions on input values. For example, consider Equation (5):

$$dF = (24.0 \cdot \varepsilon \cdot r_{pq}) \left(2.0 \cdot \frac{\sigma^{12}}{r_{s_{pq}}^7} - \frac{\sigma^6}{r_{s_{pq}}^4} \right), \text{ where } r_{s_{pq}} > 0, \sigma \in [10^{-2}, 10^{-1}], \varepsilon \in [10^{-14}, 10^{-13}]. \quad (5)$$

In the case without value-range restrictions, the analysis found an improvement of $34.0 \mapsto 15.6$ bits. However, this theoretical potential is not reasonable when considering actual variable ranges, since the algorithm checks the whole domain of input values instead of optimizing over a small feasible interval only. Consequently, only one of the analyzed expressions yielded an actual improvement after accounting for additional constraints using range annotations (e.g., parameters with constant values) obtained from PPME's code analysis. As a consequence, the analysis did not find significant improvements.[2] Hence, additional information about variables may be required to generate reliable results. A DSL like ours may help extract such information automatically.

6.3 Impact on Runtime Performance

Since the optimization modifies expressions and, in some cases, replaces a simple assignment with a complex one containing several conditional branches, its influence on runtime performance might be of concern. Therefore, we investigate the impact on execution time for the GS and LJ case studies. We compare the runtime of the original program for each use case with the optimized versions. To factor out disk-I/O from the measurements, the simulations are modified so

[2]In fact, the analysis increased the error from $5 \mapsto 11$ ($5 \mapsto 20$) bits on Racket version 6.4 (6.7) using seeds 2808995595, 415209655, 1218262282, 3135925998, 2713258581, 1066853958, and Herbie commit hash `f6ebaea`. In contrast, in version 1.0 of the tool, input and output error remained at around 4bits.

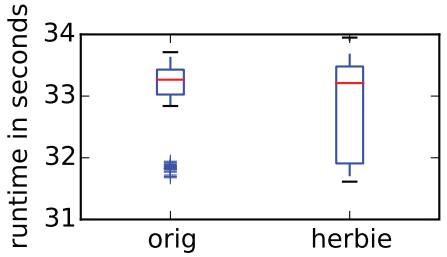


Fig. 25. Runtime comparison for the Gray-Scott example with $t_{\text{end}} = 2{,}000$. The median runtime for both simulations is nearly identical, which indicates that Herbie's optimization have no impact on the program's runtime performance.

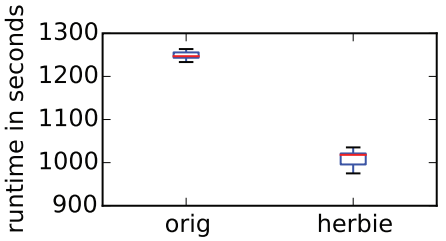


Fig. 26. Runtime comparison for the Lennard-Jones example. The execution of the program modified by Herbie was approximately 20% faster than the original implementation.

that no output is generated. The tests were run on a system with an Intel Core i3-4160 CPU, 16GB random-access memory and Ubuntu Linux with kernel 4.2.0.

We executed the GS use case 100 times per variant with 4,000 steps ($t_{start} = 0.0$, $t_{end} = 2{,}000.0$, $\Delta t = 0.5$). Figure 25 shows the variation of the execution times as a box plot. The median of both variants is nearly identical while the data are less scattered for the original simulation with a few outliers at approximately the minimal execution time of the modified program. For GS there is therefore no significant performance impact due to the accuracy optimizations.

In the case of LJ, we compared the runtime of the original program to a variant with several transformed expressions, including Equation (5) without range restrictions. The simulation is executed for $n = 5{,}000$ particles, end time $t_{\text{end}} = 0.2$, and $\Delta t = 1.0 \cdot 10^{-6}$ (200,000 steps). The results are summarized in Figure 26, both variants were run 25 times. The accuracy-optimized version runs nearly 20% faster than the original implementation. However, this can be attributed to over-simplifications of some of the expressions due to the missing range restrictions. Considering the actual numbers produced by the simulation, the two variants visibly differ in their results, with the optimized version being less accurate, yielding force values that are two orders of magnitude lower. This is the result of Herbie removing a complete subterm from the force equation. When taking parameter value ranges into account, only one expression could be improved through a simple restructuring of its terms. In this case, we can not detect any impact on the program's execution time, but the results remain correct.

7 EVALUATION

One of PPME's primary goals is to reduce the knowledge gap in scientific programming. This is achieved by providing domain-specific abstractions at the language level for particle-mesh simulations, based on those previously offered by PPML. This is complemented with features of a modern IDE, such as code completion and syntax highlighting, guiding the scientific programmer using domain-specific notations that are free of the overhead otherwise introduced by parallel programming. In addition, the formal type system and its optional extensions considerably improve error detection. They prevent a series of common errors at development-time and provide developers with meaningful feedback. Errors are captured and reported at the DSL level, instead of the level of the generated code. In comparison with PPML, PPME improves error detection and handling of the following kinds:

	PPME	PPME*	PPML	Fortran
Gray-Scott (complete)	46	28	53	623
Gray-Scott (RHS)	4	4	9	103
Lennard-Jones (complete)	88	72	75	480
N-Body (complete)	69	51	58	469

PPME*: Written Lines of Code (WLOC)

Fig. 27. Comparison of lines of code for PPME, PPML, and PPM/Fortran.

— *PPM instantiation errors.* PPME inherently generates statements in the order that is expected by the PPM call protocols. Frequently, such errors would otherwise only be discovered at runtime.
— *PPML redundant redeclaration errors.* PPML requires a proper redeclaration of fields and operators that are accessed in loops or right-hand-side specifications. A missing or wrong declaration leads to compile-time or runtime errors. Since PPME analyzes the code to derive the required information, such errors become impossible.
— *Syntax errors in Fortran code.* Except for explicitly inlined Fortran code, PPME has its own expression language so that syntactic errors in Fortran expressions, as in PPML, are impossible.
— *Type errors in Fortran code.* PPME has its own type checker so that static type errors in the generated Fortran code are not possible.
— *Dimension-related errors.* Due to dimension support, PPME is capable of statically detecting errors in expressions and differential equations. If not detected, then such errors can silently corrupt the simulation result, wasting HPC resources.

Besides error detection, PPME achieves program sizes similar to PPML or even smaller. Figure 27 compares the source lines of code (SLOC) of the implemented case studies for PPME and the generated PPML and Fortran codes. Since PPME is not a conventional editor but a projectional one, we use *written lines of code (WLOC)* as a second metric in column *PPME**. This takes into account that PPME programs typically contain some lines that have not been entered by the developer but are generated by the editor (e.g., optional configuration fields). Considering SLOC, in the Lennard-Jones and N-body examples, the PPME programs are larger than the generated PPML code with ratios of 88 : 75 (117 %) and 69 : 58 (119 %). In contrast, in the Gray-Scott example, PPME requires less space with a ratio of 50 : 62 (81 %). The reduction in the latter case is due to the built-in constructs for solving PDEs, which are not used in the other examples. Considering WLOC, PPME reduces the code sizes in all examples. In the Lennard-Jones and N-body examples, the corresponding ratios versus PPML go down to 96% and 88%, respectively. For Gray-Scott, the code size goes down to 52%.

In terms of performance, there is no difference between the execution time of code generated from PPME and an equivalent hand-written PPML version. This is due to the fact that the PPML output code generated from PPME is identical to the hand-written PPML code. Any performance difference between PPML and plain Fortran linked against the PPM Library was found to be less that 2% in previous experiments (Awile et al. 2010), whereas PPM Fortran was found to generally perform better than hand-parallelized Fortran code that does not use the PPM Library (Sbalzarini et al. 2006).

PPME also has some drawbacks. In comparison to PPML and other programming languages, version control turns out to be more complicated. Since files are serialized using XML, conventional text-based diff and merge operations are difficult to apply. While MPS has built-in support for most

of the established version-control systems, the resulting workflow is different from the text-based approaches and not always as efficient. For instance, even if the rendered program did not visibly change, or only a small edit was applied, serialization may change a lot, causing more merge conflicts in collaborative development scenarios.

Another difference between PPME and general-purpose languages is that convenience support for user-defined functions and types is not built into the PPML language. For the development of our current application examples, it was not necessary to have these concepts in the language. However, this may change in the future, demanded by more complex applications. Language extension is one of the key features of language-oriented programming in PPME/MPS, which was, in turn, not the case for PPML.

8 RELATED WORK

An exhaustive overview of language workbenches, their features, and use is given in Erdweg et al. (2013). Here, we therefore only discuss a small selection of well-known workbenches for textual languages.

Spoofax (Kats and Visser 2010) is a language workbench that builds upon term rewriting with *Stratego* (Bravenboer et al. 2008), a high-level grammar language as well as meta languages for name and type analysis. DSLs implemented in Spoofax can be used via generated plugins for the Eclipse platform or from command line. Other well-known workbenches for textual DSLs with similar capabilities on the basis of the Eclipse Modeling Framework (EMF) are Xtext (Eysholdt and Behrens 2010) and EMFText (Heidenreich et al. 2009). These tools leverage the relation of context-free grammars and the EMF (cf. Alanen and Porres (2003)) to make grammarware available to the field of model-driven software engineering. Xtext provides built-in languages for code generation and semantic functions. Furthermore, an Xtext language for formal specifications of type systems has been developed (Bettini 2015). EMFText, in contrast, follows a *convention-over-configuration* approach, which tries to provide most language features out of the box. In case that this is not sufficient to realize all intended behavior, model-based attribute grammars or the object-constraint languages (Bürger et al. 2011; Heidenreich et al. 2013) are available.

Since MPS implements a projectional editing approach (Feiler and Medina-Mora 1981; Voelter 2013), no parsers or grammars are involved, as nodes are added, deleted, and modified directly. While projectional editing is more restrictive than direct text manipulation, it is less prone to errors and serializes models as data structures, i.e., when a model is saved and loaded again, the exact instance is restored. MPS has already proven its applicability in other domains. The *mbeddr* project instantiates MPS in the embedded domain providing a projectional C frontend and several extensions and domain-specific analyses, such as state machines and model checking (Voelter et al. 2013). Moreover, the authors of Benson and Campagne (2015) used MPS to create a language and editor for automated statistic analyses of biological data (bio markers), which is designed for end-user programming and statistical visualization.

Besides language workbenches and greenfield DSL development, another possibility is to hook into already existing extensible compiler infrastructures, e.g., relying on their basic intermediate representations, default optimizations, and code generation facilities. Well-known examples for these infrastructures are the LLVM framework (Lattner and Adve 2004) and Graal (Duboscq et al. 2013). LLVM is used as a compiler backend for various general-purpose languages, most notably C and C++. It is centered around a universal intermediate language that is transformed through several extensible phases, as well as various optimizations, allocations, and code selection, down to platform-specific machine code. A DSL could rely on this infrastructure by generating code in the LLVM intermediate language, reusing and extending compiler facilities. Graal is an extensible just-in-time compiler for the Java Virtual Machine and a platform for testing new high-level

optimizations. Further, it provides support for integrating with new languages, language features, and domain-specific optimizations (Wimmer 2015).

During recent years, the importance of DSLs for scientific computing has been increasingly realized. This led to the emergence of a number of approaches of which we mention a few notable examples. Blitz++ (Veldhuizen 2000) is a template-based library and DSL for generating finite-difference operators (stencils) from high-level mathematical specifications. Freefem++ (Hecht 2012) is a software toolset and DSL for finite-element methods. This DSL allows users to define analytic as well as finite-element functions using domain abstractions such as meshes and differential operators. Liszt (DeVito et al. 2011) extends Scala with domain-specific statements for defining solvers for partial differential equations on unstructured meshes with support for parallelism through MPI, pthreads, and CUDA. The FEniCS project (Logg et al. 2012) comprises a finite element library, the unified form language (UFL) (Alnæs et al. 2014), and several optimizing compilers for generating code that can be used with the library. Building upon FEniCS, the Firedrake project (Rathgeber et al. 2015) adds composing abstractions such as parallel loop operations.

The idea of transforming or rewriting program code for optimization purposes is not new. For example, a DSL optimizer could be implemented using program transformations (Schordan and Quinlan 2003) or rewrite rules. However, research on using graph-rewrite systems for such tasks (Aßmann 2000; Schösser and Geiß 2008) indicates that the pattern language must be powerful enough to express context-sensitive patterns. Furthermore, recent research shows that rewriting is a convenient technique for implementing high-level optimizations on a restricted set of language constructs. For instance, authors in Panchekha et al. (2015) propose a method for automatically improving the accuracy of floating-point expressions by rewriting such expressions according to a set of harvested patterns. Further, the authors of Steuwer et al. (2015) apply rewriting to specific functional expressions for parallel computations to obtain efficient GPU kernels. In the field of DSLs for scientific computing, domain-specific optimizations carry great potential, since scientific codes often induce specific boundaries on data access and numeric algorithms. In Ølgaard and Wells (2010), the authors discuss different optimization strategies on representation code for element tensors in the finite-element method. The representation code is written in UFL, a high-level mathematical DSL for variational forms. The proposed strategies yield significant runtime speedups and leverage domain knowledge to automate nontrivial optimizations that normally would have been developed manually by scientific programmers. Related to that, the authors of Luporini et al. (2016) discuss loop-level optimizations for finite-element solvers in the COmpiler For Fast Expression Evaluation (COFFEE) (Luporini et al. 2015). Heuristics are used to predict local minima of operation counts at runtime, using semantics-preserving transformations such as code motion, expansion, and factorizations. The authors show that their domain-specific optimizations are superior to those that are generally applied by standard compilers such as Intel's ICC for optimizing the operation count in nested loops. Similar optimizations could be provided as extensions of PPME.

Also, the idea of adding dimensions or physical units to DSLs is not new. Cook et al. (2006) presented an analysis technique checking correctness of units in programs without extending the base language, aiming for a minimal effort of annotations for a developer. Furthermore, Austin (2006) proposed unit annotations for linear-algebra and finite-element calculations, which are similar to the dimension annotations in PPME. However, adding units to programming languages frequently has flaws. For instance, frameworks may use abstractions with boxing and unboxing of quantities and units, which implies a runtime overhead. In our approach, the analysis is optional and does not have an impact on runtime performance, since it is only used at compile-time for consistency checking and does not persist in the simulation.

9 CONCLUSIONS

We have presented the Parallel Particle-Mesh Environment (PPME), an adaptable and extensible programming environment environment with a domain-specific language for particle-mesh methods. It aims to simplify the development of scientific simulations through domain-specific abstractions and automatic generation of client code that links with the PPM library. Leveraging the Meta Programming System (MPS), we earned features that are typical for modern development environments (e.g., syntax highlighting, automatic code completion, etc.) and also features that are due to the methodology of projectional editing (e.g., mathematical notation). Furthermore, MPS provided us with a type-system language and a powerful concept of arranging our environment in a modular way, which also is one of the key enablers for extensible language design and implementation, one of the major goals of PPME.

We demonstrated PPME's capabilities in that respect in two ways: First, we developed a dimensional calculus on top of the original type system, including an extension for checking and declaring dimensions, or even measurement units. Errors discovered by the type system and the dimensional analysis are instantaneously reported to the user at design time. Second, we added support for automatic accuracy improvements of floating-point expressions by adopting Herbie as an external tool and integrating additional value-range specifications into the language. Since both extensions are designed as independent plug-in solutions, they do not interfere with the base language and the framework can easily be adapted to other cases, and developers are free to use the extensions only if desired.

Despite the obvious advantages of PPME, there are some obstacles that derive from MPS' basic principles of projectional editing and modular language specification. Due to the complexity of MPS, it is not easy for scientific programmers to develop own extensions for PPME (e.g., for loading data from a specific type of file). They have to become familiar with the concepts of MPS, which re-iterates the problem of the knowledge gap in scientific programming. PPME therefore allows the user to include custom Fortran code as inline blocks. While strictly speaking this is a design breach in a non-embedded DSL, it offers a pragmatic solution. Other issues with MPS come from projectional editing itself, which leaves less freedom than text editing w.r.t. writing comments or incomplete intermediate code. However, after some training, developers normally get used to the tool (cf. Voelter et al. (2013)). Another potential source of problems is that programs are not stored as plain text, so that using version control outside of PPME/MPS is difficult.

In the future, we will extend PPME to support more particle and mesh abstractions, including inter-particle *connections*, *neighbor lists*, and *meshes* of different topology. We also want to expand PPME to better support high-level parallelization constructs and analyses to further improve code generation and runtime scalability by leveraging the domain knowledge for more intelligent mapping and distribution of computations onto underlying parallel hardware. Finally, we will improve PPME's code generation process by adding another layer of abstraction to better integrate the target language and make the backend exchangeable. This way, we will be able to support the successor of the PPM library, OpenFPM, which is currently developed in C++.

REFERENCES

Yaser Afshar and Ivo F. Sbalzarini. 2016. A parallel distributed-memory particle method enables acquisition-rate segmentation of large fluorescence microscopy images. *PLoS ONE* 11, 4 (2016). DOI: http://dx.doi.org/10.1371/journal.pone.0152528

Marcus Alanen and Ivan Porres. 2003. *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels.* Technical report. TUCS Turku Center for Computer Science, Åbo Akademi University.

Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.* 40, 2 (2014), 9:1–9:37. DOI: http://dx.doi.org/10.1145/2566630

Uwe Aßmann. 2000. Graph rewrite systems for program optimization. *ACM TOPLAS* 22, 4 (2000), 583–637. DOI: http://dx.doi.org/10.1145/363911.363914

Mark A. Austin. 2006. Matrix and finite element stack machines for structural engineering computations with units. *Adv. Eng. Softw.* 37, 8 (2006), 544–559. DOI: http://dx.doi.org/10.1016/j.advengsoft.2005.10.004

Omar Awile. 2013. *A Domain-Specific Language and Scalable Middleware for Particle-Mesh Simulations on Heterogeneous Parallel Computers*. PhD Thesis, Diss. ETH No. 20959. ETH Zürich.

Omar Awile, Ömer Demirel, and Ivo F. Sbalzarini. 2010. Toward an object-oriented core of the PPM library. In *Proceedings of ICNAAM*. AIP, 1313–1316.

Omar Awile, Milan Mitrović, Sylvain Reboux, and Ivo F. Sbalzarini. 2013. A domain-specific programming language for particle simulations on distributed-memory parallel computers. In *Proceedings of PARTICLES*. Stuttgart, 436–447.

Josh Barnes and Piet Hut. 1986. A hierarchical O(Nlog$N$) force-calculation algorithm. *Nature* 324 (1986), 446–449.

T. Belytschko, Y. Y. Lu, and L. Gu. 1994. Element-free galerkin methods. *J. Num. Meth. Eng.* 37 (1994), 229–256. DOI: http://dx.doi.org/10.1002/nme.1620370205

Victoria M. Benson and Fabien Campagne. 2015. Language workbench user interfaces for data analysis. *PeerJ* 3, e800 (2015). DOI: http://dx.doi.org/10.7717/peerj.800

Lorenzo Bettini. 2015. Implementing type systems for the IDE with Xsemantics. *Log. Algebr. Meth. Prog.* (2015). DOI: http://dx.doi.org/10.1016/j.jlamp.2015.11.005

George C. Bourantas, Bevan L. Cheeseman, Rajesh Ramaswamy, and Ivo F. Sbalzarini. 2016. Using DC PSE operator discretization in eulerian meshless collocation methods improves their robustness in complex geometries. *Comput. Fluids* 136 (2016), 285–300.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comp. Prog.* 72, 1–2 (2008), 52–70.

David S. Broomhead and David Lowe. 1988. *Radial Basis Functions, Multi-variable Functional Interpolation and Adaptive Networks*. Technical Report. DTIC Document.

Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann. 2011. Reference attribute grammars for metamodel semantics. In *Proceedings of the SLE 2010 (LNCS)*, Vol. 6563. Springer, 22–41. DOI: http://dx.doi.org/10.1007/978-3-642-19440-5_3

Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. 1986. A simple applicative language: Mini-ML. *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (1986), 13–27. DOI: http://dx.doi.org/10.1145/319838.319847

Robert F. Cmelik and Narain H. Gehani. 1988. Dimensional analysis with C++. *IEEE Software* 5, May (1988), 21–27.

Phil Cook, Colin Fidge, and David Hemer. 2006. Well-measuring programs. In *Proceedings of ASWEC'06*, Vol. 54. IEEE, Sydney, 253–261. DOI: http://dx.doi.org/10.1016/S0261-5177(02)00005-5

P. Degond and S. Mas-Gallic. 1989. The weighted particle method for convection-diffusion equations. Part 1: The case of an isotropic viscosity. *Math. Comput.* 53, 188 (1989), 485–507. DOI: http://dx.doi.org/10.1090/S0025-5718-1989-0983559-9

Zachary De Vito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, and others. 2011. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of SC'11*. ACM, 9. DOI: http://dx.doi.org/10.1145/2063384.2063396

Sergey Dmitriev. 2004. Language oriented programming: The next programming paradigm. *JetBrains onBoard* November (2004). Retrieved from http://ranger.uta.edu/.

Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of VMIL'13*. ACM, 1–10.

Jeff D. Eldredge, Anthony Leonard, and Tim Colonius. 2002. A general deterministic treatment of derivatives in particle methods. *J. Comput. Phys.* 180 (2002), 686–709.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The state of the art in language workbenches. In *Proceedings of SLE'13*. Springer, 197–217. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11

Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement your language faster than the quick and dirty way. In *Conference Companion of OOPSLA*. ACM, 307–309.

Peter H. Feiler and Raul Medina-Mora. 1981. An incremental programming environment. In *Proceedings of ICSE'81*. IEEE Press, 44–53.

Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? Retrieved from http://martinfowler.com/articles/languageWorkbench.html.

Daan Frenkel and Berend Smit. 2001. *Understanding Molecular Simulation: From Algorithms to Applications* (2nd ed.). Elsevier, Burlington, MA. 661 pages.

P. Gray and S. K. Scott. 1983. Autocatalytic reactions in the isothermal, continuous stirred tank reactor. *Chemical Engineering Science* 38, 1 (1983), 29–43.

L. Greengard and V. Rokhlin. 1987. A fast algorithm for particle simulations. *J. Comput. Phys.* 73 (1987), 325–348.

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 1–8.

Ian J. Hayes and Brendan P. Mahony. 1995. Using units of measurement in formal specifications. *Formal Aspects Comput.* 7, 3 (1995), 329–347.

Frederic Hecht. 2012. New development in freefem++. *J. Num. Math.* 20, 3–4 (2012), 251–266. DOI: http://dx.doi.org/10.1515/jnum-2012-0013

Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. 2009. Derivation and refinement of textual syntax for models. In *Proceedings of ECMDA-FA'09 (LNCS)*, Vol. 5562. Springer, 114–129.

Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. 2013. Model-based language engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*. LNCS, Vol. 7680. Springer, 322–345. DOI: http://dx.doi.org/10.1007/978-3-642-35992-7_9

R. W. Hockney and J. W. Eastwood. 1988. *Computer Simulation Using Particles*. Institute of Physics Publishing.

Arnault Ioualalen and Matthieu Martel. 2012. A new abstract domain for the representation of mathematically equivalent expressions. *Static Analysis* (2012). DOI: http://dx.doi.org/10.1007/978-3-642-33125-1_8

Sven Karol, Pietro Incardona, Yaser Afshar, Ivo F. Sbalzarini, and Jeronimo Castrillon. 2015. Towards a next-generation parallel particle-mesh language. In *Proceedings of DSLDI'15*. 15–18.

Michael Karr and David B. Loveman. 1978. Incorporation of units into programming languages. *Commun. ACM* 21, 5 (1978), 385–391.

Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench. rules for declarative specification of languages and IDEs. In *Proceedings of OOPSLA'10*. ACM, 444–463.

Andrew Kennedy. 1994. Dimension types. *Esop* 788 (1994), 348–362.

Andrew J. Kennedy. 1997. Relational parametricity and units of measure. *Proceedings of POPL'97*, 1, January (1997), 442–455.

Monica S. Lam and Michael E. Wolf. 1991. A data locality optimizing algorithm. In *Proceedings of PLDI'91*, Vol. 39. ACM. DOI: http://dx.doi.org/10.1145/989393.989437

Peter Lancaster and Kes Salkauskas. 1981. Surfaces generated by moving least squares methods. *Math. Comput.* 37, 155 (1981), 141–158. DOI: http://dx.doi.org/10.1090/S0025-5718-1981-0616367-1

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO'04*. IEEE, 75–.

Wing Kam Liu, Sukky Jun, and Yi Fei Zhang. 1995. Reproducing kernel particle methods. *J. Num. Meth. Fl.* 20, 8–9 (1995), 1081–1106. DOI: http://dx.doi.org/10.1002/fld.1650200824

Anders Logg, Kent-Andre Mardal, and Garth Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method* (1st ed.). Number 84 in LNCSE. Springer. DOI: http://dx.doi.org/10.1007/978-3-642-23099-8

L. B. Lucy. 1977. A numerical approach to the testing of the fission hypothesis. *Astron. J.* 82 (1977), 1013–1024. DOI: http://dx.doi.org/10.1086/112164

Fabio Luporini, David A. Ham, and Paul H. J. Kelly. 2016. *An Algorithm for the Optimization of Finite Element Integration Loops*. Technical Report.

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Trans. Archit. Code Optim.* 11, 4 (2015), 57:1–57:25. DOI: http://dx.doi.org/10.1145/2687415

Leon Moonen. 2001. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering 2001*. IEEE Computer Society, 13–22. DOI: http://dx.doi.org/10.1109/WCRE.2001.957806

MPS-3.2-Documentation. 2015a. TextGen. (2015). Retrieved from https://confluence.jetbrains.com/display/MPSD32/TextGen.

MPS-3.2-Documentation. 2015b. User's Guide. (2015). Retrieved from https://confluence.jetbrains.com/display/MPSD32/MPS+User's+Guide.

nvidia. 2015. CUDA C Programming Guide v7.0.

Kristian B. Ølgaard and Garth N. Wells. 2010. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.* 37, 1 (2010), 8:1–8:23. DOI: http://dx.doi.org/10.1145/1644001.1644009

OPENACC-STANDARD.ORG. 2012. The OpenACC Application Programming Interface Version 2.0a. (2012).

OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface Version 4.0. (2013).

Pavel Panchekha, Alex Sachez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of PLDI'15*, Vol. 50. ACM, 1–11. DOI: http://dx.doi.org/10.1145/2813885.2737959

Gordon D. Plotkin. 1981. A structural approach to operational semantics. Technical report DAIMI FN-19 (1981). DOI: http://dx.doi.org/673965.html

Gordon D. Plotkin. 2004. The origins of structural operational semantics. *J. Logic Algebra. Program.* 60-61, Suppl. (2004), 3–15. DOI: http://dx.doi.org/10.1016/j.jlap.2004.03.009

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2015. Firedrake: Automating the finite element method by composing abstractions. *arXiv preprint arXiv:1501.01809* (2015).

Sylvain Reboux, Birte Schrader, and Ivo F. Sbalzarini. 2012. A self-organizing lagrangian particle method for adaptive-resolution advection–diffusion simulations. *J. Comput. Phys.* 231 (2012), 3623–3646.

Ivo F. Sbalzarini. 2010. Abstractions and middleware for petascale computing and beyond. *Intl. J. Distr. Syst. Technol.* 1(2) (2010), 40–56.

Ivo F. Sbalzarini. 2013. Modeling and simulation of biological systems from image data. *Bioessays* 35, 5 (May 2013), 482–490.

Ivo F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos. 2006. PPM—A highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comput. Phys.* 215, 2 (2006), 566–588. DOI: http://dx.doi.org/10.1016/j.jcp.2005.11.017

Markus Schordan and Dan Quinlan. 2003. A source-to-source architecture for user-defined optimizations. In *Proceedings of JMLC 2003 (LNCS)*, Vol. 2789. Springer, 214–223. DOI: http://dx.doi.org/10.1007/978-3-540-45213-3_27

Birte Schrader, Sylvain Reboux, and Ivo F. Sbalzarini. 2010. Discretization correction of general integral PSE operators in particle methods. *J. Comput. Phys.* 229 (2010), 4159–4182.

Birte Schrader, Sylvain Reboux, and Ivo F. Sbalzarini. 2012. Choosing the best kernel: Performance models for diffusion operators in particle methods. *SIAM J. Sci. Comput.* 34, 3 (2012), A1607–A1634.

Andreas Schösser and Rubino Geiß. 2008. Graph Rewriting for Hardware Dependent Program Optimizations. In *Proceedings of AGTIVE 2007 (LNCS)*, Vol. 5088. Springer, 233–248. DOI: http://dx.doi.org/10.1007/978-3-540-89020-1_17

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance openCL code. In *Proceedings of ICFP 2015*. ACM, 205–217. DOI: http://dx.doi.org/10.1145/2784731.2784754

The MPI Forum. 2012. MPI: A Message-Passing Interface Standard, Version 3.0. (2012).

Alan Mathison Turing. 1952. The chemical basis of morphogenesis. *Philos. Trans. Roy. Soc. London B: Biol. Sci.* 237, 641 (1952), 37–72.

Zerksis D. Umrigar. 1994. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices* 29, 9 (sep 1994), 135–139.

Todd L. Veldhuizen. 2000. Blitz++: The library that thinks it is a compiler. In *Advances in Software Tools for Scientific Computing*. Number 10 in LNCSE. Springer, 57–87. DOI: http://dx.doi.org/10.1007/978-3-642-57172-5_2

Markus Voelter. 2013. Language and IDE modularization and composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV*, Vol. 7680. Springer, 383–430.

Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. 2013. mbeddr: Instantiating a language workbench in the embedded software domain. *Auto. Softw. Eng.* 20, 3 (2013), 339–390. DOI: http://dx.doi.org/10.1007/s10515-013-0120-4

Mitchell Wand and Patrick O'Keefe. 1991. Automatic dimensional inference. *Computational Logic—Essays in Honor of Alan Robinson* (1991), 479–483.

Martin P. Ward. 1994. Language oriented programming. *Softw. Concepts Tools* 15, 4 (1994), 147–161. DOI: http://dx.doi.org/10.1.1.35.6369

Gregory V. Wilson. 2006. Where's the real bottleneck in scientific computing? *Amer. Sci.* 94, 1 (2006), 5.

Christian Wimmer. 2015. Graal—Tutorial at CGO'15.

Received December 2016; revised August 2017; accepted December 2017