Domain Specific Language for Materials Modeling

Ivan Kondov[0000−0002−8794−616X], Rodrigo Cortés Mejía[0000−0003−4358−9567], Marvin Müller, Nikolai Pfisterer, and Sruthy Sreenivasan

Scientific Computing Center, Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany
{ivan.kondov,rodrigo.mejia}@kit.edu

Abstract. Scientific workflow is the established method to manage models and data analyses in computational materials science consisting of many interrelated steps (jobs) running on different computing resources. This approach improves the reproducibility of such models, as well as the provenance and reusability of the data from all steps. However, the practical usability and productivity of this approach are not satisfactory and, therefore, the willingness to adopt workflow management systems in practice is still limited. We suggest a solution for this issue pursuing a model-driven engineering strategy, which includes a domain specific language and a platform to make the workflow management and the workload management systems fully transparent. On the particular example of modeling of catalysts for oxygen reduction reaction, we compare critically different aspects of two approaches: based on a traditional bare workflow management system and on the newly proposed textM language, and its supporting tools. We find that the proposed approach introduces substantial improvements and benefits over the traditional one and anticipate production deployment of textM, for example in virtual materials design projects.

Keywords: domain-specific language · model-driven engineering · computational materials science · atomic and molecular modeling

1 Introduction

The FAIR principles for scientific data [37] have been originally introduced for experimental data but they are equally important for data produced in computer simulations. In many domains of science and engineering, the models (physics-based, data-driven, or combined) consist of multiple steps organized in scientific workflows. A huge variety of workflow management systems is available for various computing architectures, platforms and scientific domains [3], and many of them can generally satisfy the FAIR principles (see Ref. [8] and references therein). The different workflow management systems offer various user interfaces: application programming interface (API), graphical user interface (GUI) or command-line interface (CLI). For example, the FireWorks [10] and Jobflow [26] workflow systems provide Python APIs for general purpose workflows. Based on these, Atomate [18] provides a Python API for managing materials simulation

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

and data analysis workflows. Nevertheless, workflow management systems still have some unsolved issues. Some technical challenges, such as materials modeling ontology, have been discussed recently in the context of materials design [28].

In this work, we address issues of non-functional nature that we have faced over the last 15 years of experience with several workflow management systems in multiple collaboration projects. Specifically, these issues are the *low usability* of workflow management tools and the *low scientist productivity* from using workflow management tools. These issues limit the general acceptance and willingness to use these tools within the materials research community. There are different causes of these issues: for instance, the great number of workflow systems with large variety of technical options can overwhelm the domain scientist in the choice of the right tool; another issue occurs when scientists need to transition to another workflow tool to address a new problem, not only because of lacking interoperability but also due to the long time required to learn how to employ such tools. This issue can be alleviated by using workflow languages, such as CWL [2], and generative tools [24,25]. However, the domain scientist still has to acquire knowledge that is far from their application domain, e.g. about managing workflow graphs and capturing provenance metadata, or resolving race conditions and deadlocks. A good workflow management system performs these actions automatically but still requires thorough understanding from the side of the user. Even after the knowledge and skill barriers have been surpassed, the technical complexity of workflow modeling causes high development effort with every new model since prototyping is very difficult and slow.

In this work, we propose a new alternative approach based on a domain-specific language (DSL), which maintains all the benefits from workflow management systems. The language allows creating, using and reusing FAIR-compliant models with low effort and without the aforementioned complex technicalities of scientific workflows. Moreover, the language provides an additional level of abstraction which conceals some of the intricacies of high performance computing, such as the resource management system and the application runtime environment.

In the next Sect. 2, related recent work will be briefly reviewed. In Sect. 3, we will outline the key concepts and elements of the textS and textM languages. Further in Sect. 4, we will demonstrate the benefits of the new textM language in a real-life use case by comparing to the more traditional scientific workflow approach applied to the same use case. In Sect. 5 we will summarize the paper.

2 Related Work

Domain-specific languages (DSLs) have been applied in different domains. Some notable examples are SQL, HTML and PostScript. In the domain of computer aided engineering, the Modelica language [9] has been successful. DSLs are terse and more expressive in their domains, and require less effort from their users because they include domain concepts and idioms which would otherwise require

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

significant effort, and result in code redundancies, if implemented using general-purpose languages such as C, C++, Python or Java. Given that the domain knowledge is directly integrated in a DSL and its supporting tools (parsers, editors, compilers, interpreters, etc.), a domain scientist learns a DSL faster and uses a DSL more productively than a general-purpose language.

As with any formal computer language, creating a DSL from scratch is a sophisticated task. This is mainly because a parser for the new language with a specific output language must be generated from a grammar by a parser generator. This process can be enormously facilitated by model-driven engineering, which has been shown to bring benefits in different domains, e.g. high-performance computing and molecular modeling [5,4,21,23]. Using the model-driven approach, creating DSLs has become a fairly feasible task [36]. For example, Xtext from the Eclipse Modeling Framework [30] has been employed to create domain-specific languages for rapid workflow development [23] and to generate high-performance computing code from algorithmic skeletons [38]. Inspired by Xtext, which has Java as output language, a similar system for Python named textX has been developed and demonstrated with the Kronos language for task scheduling [29]. Currently, twelve different projects have employed textX for developing DSLs in different domains. So far, we are not aware of any DSL for materials modeling.

Native support of physical units is provided in other languages, such as F# and Mathematica. The Unum package adds semantics for physical units to Python. The DSL workbench MontiCore can be viewed as an alternative to textX and provides a grammar for SI units.

3 The textS/textM Language Family

In this section, we introduce our domain concepts, such as the platform and the type system, illustrate the textS and textM semantics and syntax, and briefly describe the implementation of the supporting tools.

3.1 Domain Analysis

We are not aware of a formal, and sufficiently structured, domain knowledge description that captures the abstractions, notations, and constraints in computational materials science, that can be used to develop a DSL. Therefore, we have performed a continuous *domain analysis* throughout the development of the project. These aspects will be introduced in detail in the next section.

Besides the knowledge from computational materials science, we have encountered cross-sectional aspects related to the domain of computational science. These impose requirements such as using high-performance computing (HPC) environments involving resource management systems, workflow management systems, databases, and database management systems. On the other hand, we have detected a large pool of Python software resources that are heavily used in the domain. Therefore, a family of DSLs has been conceived for the platform shown in Fig. 1, left diagram. The platform integrates three back-ends: a

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

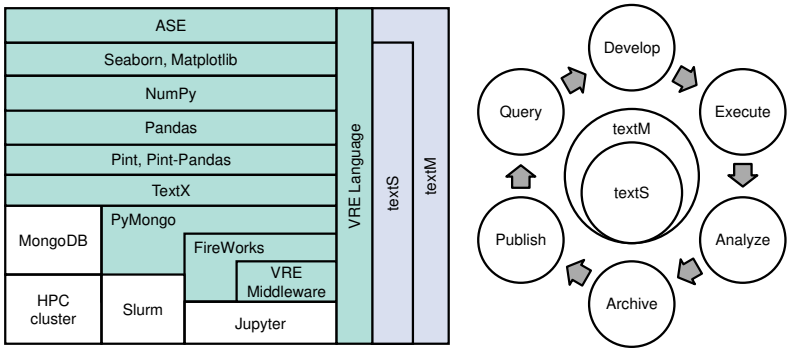

Fig. 1. Architecture stack read from the left to the right (left diagram). The Python and the domain specific components are shown on green and blue backgrounds, respectively. The full model life cycle facilitated by textS/textM (right diagram).

workflow management system, a document database, and a workload management system. Data reuse, which requires persistence of data and rich provenance metadata, is managed by the FireWorks workflow management system [10] and a MongoDB database. The computing jobs are run via Slurm on one or more HPC clusters. The back-ends are made fully transparent by the platform and the language interpreter is provided via user interfaces on two front-ends: a command-line tool to manage models in both scripting and interactive mode, and a Jupyter kernel to be used with Jupyter Notebook and Jupyter Lab. The DSL grammars, and all supporting tools have been released as open source in the VRE Language package [33].

Currently, the family of DSLs on the platform are *textS*, for general scientific computing, and *textM*, for atomic and molecular modeling, which is a proper super-set of textS. Models written in textS and textM can run on the platform and cover the entire model life cycle as shown in Fig. 1, right diagram.

3.2 Domain Concepts and Language Abstractions

The major design decision in creating a DSL for computational materials science was to depart from the notion of scientific workflow. Nonetheless, since a scientific workflow is inherent to any complex computer model in this domain, a workflow management system is still employed to store and run the model. The key difference is that its user interfaces have been concealed in the language. Considering commonly accepted practices within the materials community, the DSL relies on the declarative programming paradigm with immutable variables, rather than employing other categories such as directed acyclic graphs, workflow nodes, tasks, and links. Correspondingly, error messages are domain-specific and debugging code is mostly not needed. The major idioms in materials modeling are represented by *expressions*, which have been thoroughly considered when constructing the grammar rules. However, the *pure function* has also been integrated leaning on the work of Kelly et al. [11,12]. For example, the built-in `if` function and `if` expression are equivalent:

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

```
var1 = if(bool_expression, value_on_true, value_on_false)
var2 = value_on_true if bool_expression else value_on_false
```

Physical units of all numeric parameters, of both scalars and arrays, have been fully implemented as part of the language syntax, and are interpreted based on the Pint package [22]. The Tuple, Series, Table, and Array *data structures* can be used as stand-alone parameters, or as foundations for further domain-specific parameters of other types.

In atomic and molecular modeling the concept of atomic structure, or simply *structure*, has been established, for example in Python Materials Genomics (pymatgen) [20]. The Atomic Simulation Environment (ASE) [17] uses a similar object named `Atoms`. In contrast, the `Structure` parameter in textM does not include attributes such as `Calculator`, `Algorithm`, or `Constraint` because these are not integral parts of an atomic structure, but rather external entities. Moreover, `Structure` does not have any attributes related to these entities. Instead, the `Property` entity alone governs the relationships between these three entities and `Structure`. For example, to compute intrinsic properties of an atomic structure, such as the moments of inertia, the center of mass or the radius of gyration, no `Property` is necessary. As a rule of thumb, if at least one parameter, external to `Structure`, is necessary to compute a property, then the property must be computed via `Algorithm` or `Calculator` and retrieved from `Property`. For example, let us compute the root mean square deviation (RMSD) with respect to a reference structure `ref_struct`:

```
rmsd = Property rmsd ((algorithm: rmsd_algo), (structure: struct1))
rmsd_algo = Algorithm RMSD ((reference: ref_struct))
```

Obviously, this property can be modeled using an existing abstraction from textS, e.g. to define and then call an internal function:

```
rmsd_func(struct, reference) = ...
rmsd = rmsd_func (struct1, struct2)
```

However, the use of `Property` and `Algorithm` is more expressive in this domain and provides the necessary separation of concerns, for example if more parameters than just the reference structure have to be modeled.

3.3 Type System

One lesson learned from the experience with different workflow management systems, specifically with FireWorks, was that a static type system would be necessary to prevent errors from being deferred to run time and to improve the quality of error messages. For example, in recent studies using FireWorks [27,1,31] errors often have occurred only after consuming a significant amount of computing time. Many of such errors can be detected before the expensive evaluation is started. Therefore, every parameter in textS/textM has a type attribute that is evaluated after the model has been successfully parsed. The type in textS/textM is not annotated but inferred from types of literals and imported parameters, and types returned by certain operations, which makes the input code terse and more readable.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

The plain types in textS are String, Boolean, Integer, Float, and Complex. These types can be combined to create data structures, such as Tuple, Series, Table, and Array, that are types themselves. The types in textM include Structure, Calculator, Algorithm, Property, Constraint, Trajectory, Species, Reaction. Most of the textM types are *iterable* and can be used in the same operations as Table and Series. This facilitates enormously locating properties in these parameters by using sub-scripting and performing single-instruction – multiple-data operations, e.g. by using functions such as `map` and `filter`.

3.4 Language Implementation

The use of the textX tool [7] and the Arpeggio parser [6] has been essential for the rapid development of the textS/textM languages and their supporting tools, including a Python interpreter to integrate the three back-end systems and several domain-specific Python packages. The process involving these packages is shown in Fig. 2. First, the textX meta-language is used to define the textS and textM grammars (see Ref. [33] for more details) which are processed by the textX parser. This parser is, in turn, generated by using the higher-level textX grammar. The result from parsing the textS/textM grammar includes a meta-model, which is a set of Python classes with relationships, and an in-memory textS/textM parser. The generated textS/textM parser then parses the user input and creates the *model* as a set of plain Python objects that are instances of the textS/textM meta-model classes. These objects are then processed by object and model *processors* to apply constraints, such as to check types, detect cyclic dependencies etc. Eventually, the interpreter uses the model to create a *persistent workflow* that can be evaluated completely or partially on-demand, employing local or HPC resources by employing the VRE Middleware package [34]. The nodes of the persistent workflow contain the relevant textM statements as meta-data, so that the original user input persists as well. All processes described above, starting from parsing the grammar to running the interpreter, are fully automated [33].

Several Python packages are reused by the interpreter (see Fig. 1): ASE for atoms, calculators for various electronic structure and molecular dynamics codes, Pint for units and Pandas to implement the Table and Series data structures, Numpy to implement the Array, and Seaborn/Matplotlib for data visualization. The VRE Middleware package [34] is used to configure the HPC resources and the run-time environment of model statements evaluated as batch jobs.

3.5 Additional Features

A persistent model can be extended at any time in its life cycle (see Fig. 1, right diagram) via the same mechanism outlined in Sect. 3.4 and in Fig. 2. Simple references in new variables enable intuitive data reuse within a model. For example, to compute the RMSD for a set of atomic structures `structs` with the same reference structure `ref_struct` in Sect. 3.2, one can load the model containing these variables and add a new `Property` with a reference to the same `rmsd_algo`:

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

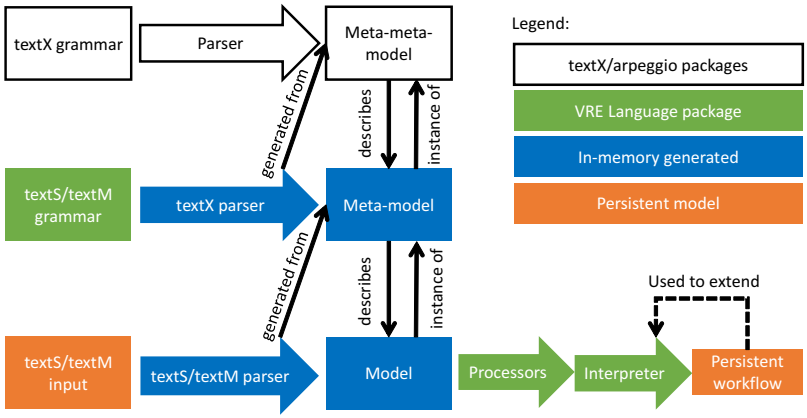

Fig. 2. DSL engineering and processing stages of a model written in textS/textM

```
rmsds = Property rmsd ((algorithm: rmsd_algo), (structure: structs))
```

If `rmsd_algo` and `structs` are in other models in the database, they and all their ancestor statements can be reused by using this syntax:

```
rmsds = Property rmsd ((algorithm: rmsd_algo@uuid_of_model1),
                       (structure: structs@uuid_of_model2))
```

Suppose that we want to compute RMSD without aligning the structure and the reference, then we can simply add:

```
vary ((rmsd_algo: Algorithm RMSD
                  ((reference: ref_struct), (adjust: false))))
```

This will create a new model, still bundled with the original model in a group, which automatically performs the evaluation of everything depending on `rmsd_algo` with the new values specified in the `vary` statement. This feature implements an idiom often called *parameter sweep*, or *parameter scan*, commonly used in high-throughput computing.

4 Use Case: Oxygen Reduction Reaction Catalyst

In this section, we perform a systematic evaluation of the textM language and demonstrate the substantial improvements and benefits of the proposed approach compared to the traditional FireWorks workflow approach. We show how the textM language is applied to a model describing catalyst materials for oxygen reduction reaction. Originally, the model has been implemented with the FireWorks workflow management system using Python and YAML and employed in several application studies [1,27,31]. This previous work allows us to make a fair comparison to the new textM-based implementation. Due to the large number of lines of Python/YAML code, a direct side-by-side comparison to textM is not possible in this paper. Instead, we compare the number of code lines and provide the complete source code of both variants in an open source repository [35].

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

4.1 Brief Model Description

The model is based on the original work of Nørskov et al. [19]. The computational workflow has been discussed in detail in Sect. 4.3 of Ref. [28] and shown in Fig. 4 therein. Here we outline the main physical quantities in order to understand the textM-based implementation.

The oxygen reduction reaction (ORR) is the electrochemical process taking place at the cathode of a fuel cell or a metal-air cell. The ORR has a mechanism consisting of four elementary reactions

$$\mathrm{M} + \mathrm{O_2} + (\mathrm{H^+} + \mathrm{e^-}) \longrightarrow \mathrm{MOOH} \tag{1}$$

$$\mathrm{MOOH} + (\mathrm{H^+} + \mathrm{e^-}) \longrightarrow \mathrm{MO} + \mathrm{H_2O} \tag{2}$$

$$\mathrm{MO} + (\mathrm{H^+} + \mathrm{e^-}) \longrightarrow \mathrm{MOH} \tag{3}$$

$$\mathrm{MOH} + (\mathrm{H^+} + \mathrm{e^-}) \longrightarrow \mathrm{M} + \mathrm{H_2O} \tag{4}$$

where the symbol M denotes an active site on the surface of the catalyst. The quantity describing the thermodynamic efficiency of the catalyst is the onset potential, i.e. the minimum electrode potential at which the ORR is activated. The difference between the standard electrode potential U_0 and the onset potential is called *overpotential*. For the $\mathrm{O_2/H_2O}$ couple, U_0 is 1.23 V versus the standard hydrogen electrode. The design goal is to increase the onset potential of the cathode for ORR. We employ the critical potential $U_{\max}$ as a descriptor that can be regarded as an upper thermodynamic bound of the ORR onset potential, and is the least upper bound of the electrode potential U at which all elementary reactions in Eqs. (1-4) are spontaneous, i.e.

$$U_{\max} = \sup\left\{U : \Delta G^{(r)}(U) \leq 0 \quad \forall r \in [1,2,3,4]\right\}. \tag{5}$$

Thus, the defined critical potential allows us to define a bound for the ORR overpotential as $\eta_{\mathrm{ORR}} = U_0 - U_{\max}$. The free energies $\Delta G^{(r)}(T,U)$ for given temperature T and potential U can be computed from the relative reaction energies $\Delta E^{(r)}$ as

$$\Delta G^{(r)}(T,U) = \Delta G_0^{(r)} - z_r U \approx \Delta E^{(r)} + \Delta \mathrm{ZPE}^{(r)} - T\Delta S^{(r)} - z_r U \ . \tag{6}$$

In Equation (6), $\Delta E^{(r)}$, $\Delta \mathrm{ZPE}^{(r)}$, $\Delta S^{(r)}$ and z_r are the potential energy, the zero-point vibrational energy, total entropy, and transferred charge of the rth reaction, respectively. The quantities $\Delta E^{(r)}$, $\Delta \mathrm{ZPE}^{(r)}$, and $\Delta S^{(r)}$ are computed using density functional theory (DFT) applied to the relevant adsorbed (M−OOH, M−O and M−OH) and gaseous ($\mathrm{H_2O}$, $\mathrm{H_2}$ and $\mathrm{O_2}$) species. Furthermore, standard conditions (pressure of 1 bar and temperature 298.15 K) are used, while each proton–electron pair ($\mathrm{H^+} + \mathrm{e^-}$) from Eqs. (1-4) is replaced by $\frac{1}{2}\mathrm{H_2}$ gas-phase molecule, which has the same free energy under these conditions [19].

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

4.2 Implementation in textM

As first step, we provide a set of tags with which we can find the model later in the database:

```
tag {metal: 'Pt', 'active site type': 'fcc', environment: 'vacuum'}
```

Once the model is added to the database, the tag can be modified at any time. One can start creating the model by defining the main quantities in Eq. (5), Eq. (6), and the reactions in Eqs. (1-4):

```
eta_orr = U_0 - U_max; U_max = min(orr_potentials)
min(s) = reduce((x, y: if(x < y, x, y)), s)
orr_potentials =  map((g: -g/(n * e)), orr_free_energies)
n = 1  # number of transferred electrons per elementary reaction
n_elec = 4  # total number of transferred electrons
e = 1 [elementary_charge]; U_0 = 1.23 [V]
orr_free_energies = map((r: r.free_energy[0]), orr)
orr = (reactions: orr1, orr2, orr3, orr4)
orr1 = Reaction M + O2 + 0.5 H2 = MOOH
orr2 = Reaction MOOH + 0.5 H2 = MO + H2O
orr3 = Reaction MO + 0.5 H2 = MOH
orr4 = Reaction MOH + 0.5 H2 = M + H2O
```

It is striking how the modeling starts with the target quantity so that the modeler does not have to think about the control flow, i.e. which step follows which. This way one can draft a new model quickly from scratch by only following the functional dependencies. Next we need to specify the species. Since all species are defined similarly, here only MOOH is shown as an example:

```
MOOH = Species MOOH ((energy: opt_OOH.energy[0]),
        (zpe: 0.5 * sum(nrm_OOH.vibrational_energies[0])),
        (entropy: vib_entr_f(nrm_OOH.vibrational_energies[0])),
        (temperature: temperature))
```

The quantities $\Delta E^{(r)}$, $\Delta \mathrm{ZPE}^{(r)}$, and $\Delta S^{(r)}$ are defined as properties of the MOOH species, and are organized in a table with columns `energy`, `zpe` and `entropy`. The free energy of MOOH is computed by textM automatically from these properties. The calculation contains calls to internal functions and requires importing some external functions:

```
vib_entr_f(v) = kB * sum(map(entr, v, map(exp_hob, v)))
kB = 1 [boltzmann_constant]; temperature = 298.15 [K]
exp_hob(ene) = exp(-ene * beta); beta = 1.0 / (kB * temperature)
entr(ene, eh) = ene * beta * eh / (1.0 - eh) - log(1.0 - eh)
use exp, log from numpy
```

Now, only two parameters are needed to calculate the MOOH species: `opt_OOH` and `nrm_OOH`. These correspond to DFT geometry optimization and normal mode calculations, both performed on 40 cores for 2 hours:

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

```
opt_OOH = Property energy, forces, dipole (
                (structure: geom_surface_OOH),
                (calculator: calc_rlx),
                (constraints: (fix_rlx_OOH))) on 40 cores for 2.0 [hours]
nrm_OOH = Property vibrational_energies, energy_minimum (
              (structure: opt_OOH.output_structure),
              (calculator: calc_nrm),
              (constraints: (fix_nrm_OOH))) on 40 cores for 2.0 [hours]
```

The input atomic structure `geom_surface_OOH` is loaded from a file in any format that can be read by ASE. The two `Calculator` parameters are defined for using the VASP code [13,14,15,16] as follows:

```
calc_rlx = Calculator vasp == 5.4.4 (...), task: local minimum
calc_nrm = Calculator vasp == 5.4.4 (...), task: normal modes
```

The ellipsis (...) denotes a set of calculator-specific parameters, omitted here for brevity, using the same table syntax as shown above for `Property` and `Species`. The remaining geometry constraints `fix_rlx_OOH` to fix the bottom layer of atoms, and `fix_nrm_OOH` to fix all metal atoms can be defined as:

```
constr(n, m) = map((x: x < n), range(0, m, 1))
c_surf_OOH = constr(9, 30); c_nrm_OOH = constr(27, 30)  # boolean series
fix_rlx_OOH = FixedAtoms where c_surf_OOH
fix_nrm_OOH = FixedAtoms where c_nrm_OOH
```

Constraints can be defined in various ways, such as using tags (if available) in the atoms section of the `Structure` parameter, setting the displacements along the z-axis, or using chemical symbols explicitly. For example, to fix all platinum atoms one can write:

```
c_nrm_OOH = map((x: x == 'Pt'), geom_surface_OOH.atoms[0].symbols)
```

4.3 Assessment of the DSL Approach

As with other non-functional requirements, providing a rigorous measurement to assess usability and productivity is no trivial task. One typical approach is to perform continuous measurements during operation and/or perform extensive user surveys. These methods require the tools to be in the production deployment phase, which unfortunately is not yet the case for the VRE Language. Therefore, as a quantitative measure for the productivity we suggest the number of written lines of code which is shown in Table 1. In addition, we discuss several evident qualitative differences that we have detected in the course of our comparison of the two approaches throughout the full life cycle of the model.

4.4 Create a Model

To use FireWorks, one has to understand the formal workflow structure of the model and draft the control flow and data flow. This yields a directed acyclic graph that is a formal representation of the workflow. Then the individual nodes

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

Table 1. Number of human-written lines of code. Commented and empty lines, and lines of the input atomic structures, and common files have not been counted.

	Python	YAML (input)	YAML (config)	textM	Bash	Total
FireWorks	537	684	20	0	15	1256
VRE Language	0	0	2	237	5	244

have to be implemented as use-case specific Python functions. At the end the workflow graph is implemented either as a Python script or as a YAML document as here. This implementation has no concepts of data types nor physical units for the numeric types; it merely provides "glue code" to ASE and FireWorks.

As shown in Sect. 4.2, to implement the model in textM, we do not need to create a workflow graph but rather write declarative statements: typically variables defining what to compute (physical quantities, objects from the domain, etc.) but not how to compute them. The interpreter infers the control flow and data flow from these statements, and automatically creates a FireWorks workflow and adds it to the database.

The savings in the number of human-written lines of code, shown in Table 1, can be divided into two categories: i) node attributes (ID, name and further metadata) and links between the nodes are not coded in textM; ii) Python code written for the specific use case is replaced by terse textM code. It should be noted that the Python code in the textM interpreter used in this specific use case is use-case agnostic and sufficiently generic to be reused in other use cases; see the examples folder in the VRE Language [33] repository. In both categories, it becomes evident that textM is more expressive, while allowing more concise code compared to Python or YAML.

4.5 Evaluate a Model

The evaluation of a model in FireWorks is performed node by node using either the CLI provided by FireWorks, or the CLI, Python API or GUI provided by the VRE Middleware package [34]. All configuration metadata such as node category (interactive or batch), worker name, required computing resources, and application run-time environment must be specified in either in the workflow description, or in the configuration files of the worker and queue adapter. This is reflected in the number of YAML lines of code in Table 1.

By using textM the available, and the default, resources and environments are described in a `resconfig.yaml` configuration file. This is achieved through a tool from the VRE Middleware package. By simply specifying `on 40 cores for 2.0 [hours]` in a statement (see Sect. 4.2), the interpreter changes the category of the created workflow node and evaluates in which queue, and with what resources, to configure the node. In addition, depending on the calculator's type, version and task, it adds environment variables and modules to the run-time configuration. The two configuration lines in `resconfig.yaml` are used to set custom names of the VASP environment module and the VASP executable, both of which can vary over different computing clusters.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

4.6 Extend a Model

Through extensions one can add further statements to a persistent model. This allows not only to perform data analysis, but also to extend the model by reusing persistent data. In both approaches extensions can be performed at any time, independent of the state of the model, i.e. whether variables have been evaluated.

In FireWorks an extension can be performed using the CLI, or Python API, after having prepared a workflow in YAML, or Python, respectively, and all the necessary Python functions for the tasks. For the extension to work, one must perform database queries to identify the IDs of the relevant parent workflow nodes, ensuring that all data-flow links are correct, and specifying them in the extension call.

In comparison, extensions in textM can be done in the same way as creating a textM model from scratch. Instead of parent IDs and data-flow links only the references to existing variables are needed. The names of the variables can be simply found in the CLI tool (`texts session`) by typing the `%history` magic command. To demonstrate this, we extend the model described in Sect. 4.2 for computing the overpotential of the oxygen evolution reaction (OER), which is the reverse process of ORR and, also, the ORR rate-determining step. This extension merely requires nine lines of additional code, also included in the total lines for textM in Table 1. Here, we illustrate the extension for the OER overpotential:

```
eta_oer = U_min - U_0; U_min = max(map((g: -g/(n*e)), oer_free_energies))
oer_free_energies = map((r: -r.free_energy[0]), orr)
```

The OER critical potential $U_{\min}$ is defined as the greatest lower bound of the electrode potential U at which all reactions defined in Eqs. (1-4) are spontaneous in the reverse direction, thus the minus sign in the reactions' free energies `oer_free_energies`.

4.7 Archive and Publish a Model

To archive, a workflow can be dumped to a file in standard JSON format. The archive can then be used in a data publication, for example Ref. [32]. One drawback of publishing the workflow is that one has to include all Python and YAML code, otherwise the workflow is no longer reproducible. Nevertheless, the JSON archive can still be employed as reference, since it includes the input and output data of all workflow steps.

In the textM realization, the number of lines of code needed to reproduce and publish the model is substantially smaller compared to the pure workflow approach. The VRE Middleware and VRE Language packages, necessary to reproduce (or reuse parts of) the textM model, are already available as open source and not needed in the publication. A JSON archive of the workflow including the produced data would still be necessary as reference.

5 Conclusion

In this work, we have introduced a novel approach to materials modeling and data analysis based on the domain-specific language textM. Compared to a con-

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

ventional approach based solely on a workflow management system, textM shows equivalent capabilities and provides more features. The proposed approach introduces benefits in terms of improved tools, usability, and developer's productivity through all stages of the model's life cycle. The textM language is not intended to replace C++ or Fortran used for implementing HPC algorithms. Instead, it provides a supplementary platform that facilitates the integration of widely used tools in composite computer models, such as domain-specific packages, data-analysis codes, HPC resources, and data life-cycle management applications. Compared to the conventional approach, the textM language has a minimal learning curve and can be particularly appealing for non-programmers and domain experts unfamiliar with functional or declarative programming paradigms.

Future work will enable calculations including quantities with uncertainties and improve: i) the grammar and interpreter modularity to ease the creation of new languages in the VRE Language family; ii) the Jupyter kernel usability by adding highlighting, completer, and magic commands; iii) the type inference by using a Hindley–Milner algorithm.

Acknowledgments. This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Disclosure of Interests. The authors have no competing interests.

References

1. Akbari, N., Kondov, I., Vandichel, M., Aleshkevych, P., Najafpour, M.M.: Oxygen-Evolution Reaction by a Palladium Foil in the Presence of Iron. Inorganic Chemistry **60**(8), 5682–5693 (Apr 2021). https://doi.org/10.1021/acs.inorgchem.0c03746
2. Amstutz, P., Crusoe, M.R., Tijanić, N., Chapman, B., Chilton, J., et al.: Common Workflow Language, v1.0 (2016). https://doi.org/10.6084/m9.figshare.3115156.v2
3. Amstutz, P., Mikheev, M., Crusoe, M.R., Tijanić, N., Lampa, S.: Existing Workflow Systems (2024), https://s.apache.org/existing-workflow-systems
4. Bender, A., Poschlad, A., Bozic, S., Kondov, I.: A service-oriented framework for integration of domain-specific data models in scientific workflows. Procedia Computer Science **18**, 1087–1096 (2013). https://doi.org/10.1016/j.procs.2013.05.274
5. Bruel, J.M., Combemale, B., Ober, I., Raynal, H.: MDE in practice for computational science. Procedia Computer Science **51**, 660–669 (2015). https://doi.org/http://dx.doi.org/10.1016/j.procs.2015.05.182
6. Dejanović, I., Milosavljević, G., Vaderna, R.: Arpeggio: A flexible PEG parser for Python. Knowledge-Based Systems **95**, 71–74 (Mar 2016). https://doi.org/10.1016/j.knosys.2015.12.004
7. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: A Python tool for Domain-Specific Languages implementation. Knowledge-Based Systems **115**, 1–4 (Jan 2017). https://doi.org/10.1016/j.knosys.2016.10.023
8. Diercks, P., Gläser, D., Lünsdorf, O., Selzer, M., Flemisch, B., et al.: Evaluation of tools for describing, reproducing and reusing scientific workflows (Aug 2023). https://doi.org/10.48694/INGGRID.3726

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

9. Fritzson, P., Pop, A., Abdelhak, K., Ashgar, A., Bachmann, B., et al.: The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. Modeling, Identification and Control: A Norwegian Research Bulletin **41**(4), 241–295 (2020). https://doi.org/10.4173/mic.2020.4.1
10. Jain, A., Ong, S.P., Chen, W., Medasani, B., Qu, X., et al.: FireWorks: A dynamic workflow system designed for high-throughput applications. Concurrency and Computation: Practice and Experience **27**(17), 5037–5059 (2015). https://doi.org/10.1002/cpe.3505
11. Kelly, P.M.: Applying Functional Programming Theory to the Design of Workflow Engines. Ph.D. thesis, School of Computer Science, The University of Adelaide (Jan 2011)
12. Kelly, P.M., Coddington, P.D., Wendelborn, A.L.: Lambda calculus as a workflow model. Concurrency and Computation: Practice and Experience **21**(16), 1999–2017 (Nov 2009). https://doi.org/10.1002/cpe.1448
13. Kresse, G., Furthmüller, J.: Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Computational Materials Science **6**(1), 15–50 (1996). https://doi.org/10.1016/0927-0256(96)00008-0
14. Kresse, G., Furthmüller, J.: Efficient iterative schemes for *ab initio* total-energy calculations using a plane-wave basis set. Phys. Rev. B **54**(16), 11169–11186 (1996). https://doi.org/10.1103/PhysRevB.54.11169
15. Kresse, G., Hafner, J.: *Ab initio* molecular dynamics for liquid metals. Phys. Rev. B **47**(1), 558–561 (1993). https://doi.org/10.1103/PhysRevB.47.558
16. Kresse, G., Hafner, J.: *Ab initio* molecular-dynamics simulation of the liquid-metal amorphous-semiconductor transition in germanium. Phys. Rev. B **49**(20), 14251–14269 (1994). https://doi.org/10.1103/PhysRevB.49.14251
17. Larsen, A.H., Mortensen, J.J., Blomqvist, J., Castelli, I.E., Christensen, R., et al.: The atomic simulation environment — a Python library for working with atoms. Journal of Physics: Condensed Matter **29**(27), 273002 (2017). https://doi.org/10.1088/1361-648X/aa680e
18. Mathew, K., Montoya, J.H., Faghaninia, A., Dwarakanath, S., Aykol, M., et al.: Atomate: A high-level interface to generate, execute, and analyze computational materials science workflows. Computational Materials Science **139**, 140–152 (2017). https://doi.org/10.1016/j.commatsci.2017.07.030
19. Nørskov, J.K., Rossmeisl, J., Logadottir, A., Lindqvist, L., Kitchin, J.R., et al.: Origin of the overpotential for oxygen reduction at a fuel-cell cathode. The Journal of Physical Chemistry B **108**(46), 17886–17892 (2004). https://doi.org/10.1021/jp047349j
20. Ong, S.P., Richards, W.D., Jain, A., Hautier, G., Kocher, M., et al.: Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis. Computational Materials Science **68**, 314–319 (2013). https://doi.org/http://dx.doi.org/10.1016/j.commatsci.2012.10.028
21. Palyart, M., Ober, I., Lugato, D., Bruel, J.M.: HPCML: A modeling language dedicated to high-performance scientific computing. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud Computing. pp. 6:1–6:6. MDHPCL '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2446224.2446230
22. Pint Authors: Pint: makes units easy (2025), https://pint.readthedocs.io, accessed on 2025-01-27
23. Rabbi, F., MacCaull, W.: T$_\square$: A domain specific language for rapid workflow development. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18

Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 7590, pp. 36–52. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_4
24. Roozmeh, M., Kondov, I.: Workflow generation with wfGenes. In: IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). pp. 9–16. Institute of Electrical and Electronics Engineers (IEEE) (2020). https://doi.org/10.1109/WORKS51914.2020.00007
25. Roozmeh, M., Kondov, I.: Automating and Scaling Task-Level Parallelism of Tightly Coupled Models via Code Generation. In: Groen, D., de Mulatier, C., Paszynski, M., Krzhizhanovskaya, V.V., Dongarra, J.J., et al. (eds.) Computational Science – ICCS 2022, vol. 13353, pp. 69–82. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-08760-8_6
26. Rosen, A.S., Gallant, M., George, J., Riebesell, J., Sahasrabuddhe, H., et al.: Jobflow: Computational Workflows Made Simple. Journal of Open Source Software **9**(93), 5995 (Jan 2024). https://doi.org/10.21105/joss.05995
27. Salmanion, M., Kondov, I., Vandichel, M., Aleshkevych, P., Najafpour, M.M.: Surprisingly Low Reactivity of Layered Manganese Oxide toward Water Oxidation in Fe/Ni-Free Electrolyte under Alkaline Conditions. Inorganic Chemistry **61**(4), 2292–2306 (Jan 2022). https://doi.org/10.1021/acs.inorgchem.1c03665
28. Schaarschmidt, J., Yuan, J., Strunk, T., Kondov, I., Huber, S.P., et al.: Workflow Engineering in Materials Design within the BATTERY 2030 + Project. Advanced Energy Materials **12**(17), 2102638 (May 2022). https://doi.org/10.1002/aenm.202102638
29. Simić, M., Boškov, N., Kaplar, A., Dejanovic, I.: Kronos: A DSL for scheduled tasks based on textX. In: Zdravković, M., Konjović, Z., Trajanović, M. (eds.) ICIST 2017 Proceedings. pp. 358–360 (2017)
30. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Eclipse Series, Addison-Wesley Professional, 2nd edition edn. (2008)
31. Vandichel, M., Laasonen, K., Kondov, I.: Oxygen Evolution and Reduction on Fe-doped NiOOH: Influence of Solvent, Dopant Position and Reaction Mechanism. Topics in Catalysis **63**(9-10), 833–845 (Sep 2020). https://doi.org/10.1007/s11244-020-01334-8
32. Vandichel, M., Laasonen, K., Kondov, I.: Oxygen evolution and reduction on Fe-doped NiOOH. Materials Cloud Archive **2022.93** (2022). https://doi.org/10.24435/materialscloud:wh-nv
33. VirtMat Tools Team: Domain-Specific Language of the Virtual Research Environment (2025), https://vre-language.readthedocs.io, accessed on 2025-01-27
34. VirtMat Tools Team: Middleware of the Virtual Research Environment (2025), https://vre-middleware.readthedocs.io, accessed on 2025-01-27
35. VirtMat Tools Team: Oxycat: High-throughput design of OER and ORR (2025), https://gitlab.kit.edu/kit/virtmat-tools/oxycat-use-case, accessed on 2025-01-27
36. Voelter, M.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform (2013)
37. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., et al.: The FAIR Guiding Principles for scientific data management and stewardship. Scientific Data **3**(1), 160018 (Mar 2016). https://doi.org/10.1038/sdata.2016.18
38. Wrede, F., Rieger, C., Kuchen, H.: Generation of high-performance code based on a domain-specific language for algorithmic skeletons. The Journal of Supercomputing **76**(7), 5098–5116 (Jul 2020). https://doi.org/10.1007/s11227-019-02825-6

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97564-6_18