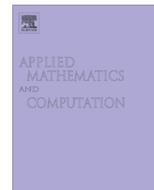
Applied Mathematics and Computation 267 (2015) 282–293

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc



# ViennaMaterials – A dedicated material library for computational science and engineering

Josef Weinbub [a,*], Matthias Wastl [a], Karl Rupp [a,b], Florian Rudolf [a], Siegfried Selberherr [a]

[a] Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, 1040 Wien, Austria
[b] Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10/E101, 1040 Wien, Austria

ARTICLE INFO

*Keywords:*
Material library
Python
C++
XML
ViennaMaterials
Units

ABSTRACT

The design and implementation aspects of the dedicated C++ materials library ViennaMaterials for science and engineering is discussed. The library's focus is to provide flexible application programming interfaces for accessing material data. Special attention is on handling physical units as well as supporting mathematical material models via a nested code evaluation mechanism. The challenges of supporting different programming languages, physical units, mathematical models, and a run-time setting are tackled by utilizing external tools on top of a flexible object-oriented library structure. Application examples covering the utilization with a numerical simulation as well as a remote network-based material database are discussed to underline the usability of ViennaMaterials.

© 2015 Elsevier Inc. All rights reserved.

## 1. Introduction

Simulation tools usually require a large set of material parameters to carry out scientific simulations, due to the use of equations which include material parameters to model the physical environment [1,2]. Among such equations, partial differential equations are especially widespread in the description of complex phenomena and they are therefore of special interest for computational science and engineering in general. Considering the vast number of phenomena and the related sets of equations for which simulation environments have been and are currently developed, it becomes apparent that many different material parameters have to be made available in a consistent and reliable manner. The challenge lies not only in the efficient storage of the material data, but also in the convenient data access. For instance, considering the case of charge carrier mobility in the field of semiconductor device simulation, in the simplest model the mobility value for a given material is a constant. However, in more complicated models, the carrier mobility depends on a set of parameters and the solution variables [3]. The storage and access methods have to support such setups.

In general, parameters can be provided by hard-coding them into the implementation, or during run-time, by, for example, feeding the parameters to the application via an input file. Where the compile-time approach is considered to be quick with respect to implementation time, the approach lacks reusability and flexibility. The parameters cannot be easily reused by different applications and changes require recompilation. On the contrary, providing the parameters during run-time allows for changing the parameters without recompilation, albeit introducing the need for additional mechanisms to enable

* Corresponding author.
*E-mail addresses:* weinbub@iue.tuwien.ac.at (J. Weinbub), wastl@iue.tuwien.ac.at (M. Wastl), rupp@iue.tuwien.ac.at (K. Rupp), rudolf@iue.tuwien.ac.at (F. Rudolf), selberherr@iue.tuwien.ac.at (S. Selberherr).

http://dx.doi.org/10.1016/j.amc.2015.03.094
0096-3003/© 2015 Elsevier Inc. All rights reserved.

data access. String-based data, such as data encoded via the extensible markup language (XML), must be potentially parsed to extract the valuable information, in turn allowing to store the data in numerical objects.

Another challenge is to incorporate unit-aware material parameters which are especially important for robust numerical simulations [4]. Not only need the units be linked to each material parameter in the data set, but also automatic unit checks and – if possible – unit conversions have to take place. Such a unit system must support the dynamic nature of the material parameter database, induced by the fact that material parameters are loaded during run-time. In other words, the unit information is usually available as a string, thus the unit system has to be able to process string-based unit expressions. Different approaches for implementing unit systems have been investigated [5–8].

Tree-based structures merit special consideration for storing material data. The underlying data associated with materials is inherently hierarchical, and can thus be mapped to a tree naturally (Fig. 1).

Albeit the fact that a tree-based structure offers flexibility and a straightforward storage of material data values, it does not natively support dependent parameters or inheritance. Considering cases where parameters are described by analytic functions, the required input dependencies must be provided externally, for instance, by simulation tools. In turn, inheritance would allow for creating specialized child materials from a reference parent material. However, inheritance is not natively supported by tree-based structures and thus needs to be added manually.

Tree-based data structures natively support path-wise data access, due to the supported intuitive descent along the structure of the tree holding the data. Languages which support a tree-based data structure and path-based data access are, for instance, XML in combination with the XML path language (XPath) as, for instance, provided by pugixml [9]. XML has a distinct advantage over in-house languages, as XML is widespread, thus parameter files can immediately use synergy effects from the large available XML ecosystem, such as graphical user interfaces (GUIs) or other well-known XML libraries, like libxml2 [10]. An alternative to XML and XPath is ViennaIPD [11]12, which has been originally designed for the field of semiconductor device simulation. ViennaIPD is a powerful control language implemented in C/C++, providing a C-like input file language, object-oriented structuring of datasets, inheritance mechanisms, and a unit system. The data stored in a ViennaIPD file is accessed and adapted from external applications via the application programming interface (API). The following snippet depicts the inheritance mechanism which can be used in input data files; a base class `A` provides data members `a` and `b`, whereas a derived class `B` inherits `a` and overwrites `b` from the base class `A`.

$$\begin{aligned} &\mathrm{A} \qquad \{\mathrm{a} = 1; \mathrm{b} = 2 * \mathrm{a};\} \\ &\mathrm{B}: \quad \mathrm{A}\,\{\mathrm{b} = 3 * \mathrm{a};\} \end{aligned}$$

Although ViennaIPD provides an elaborate mechanism to store and access data files relevant for simulations, compared to the presented approach, the language is not wide-spread and thus limits usability and applicability. However, for storing material data ViennaIPD's language is more convenient due to the absence of tags which bound scoped regions, as is being used by XML.

In the following an overview of the approaches used by some other simulation tools from different fields of science and engineering is given. The proprietary COMSOL Multiphysics suite [13] offers a material library which can be exported to an XML file or extended by importing data. Due to the closed nature of the software, utilization of the material library in other tools is not possible. However, the exported XML data can be imported in other tools, further underlining the attractiveness of XML as the primary language to store material data. The open source computational fluid dynamics toolbox OpenFOAM uses a specific language for not only storing the material parameters, but also for controlling the simulation via a file [14]. Free open source simulators from the field of semiconductor device simulation, such as ViennaSHE [15] and Archimedes [16], currently use a hard-coded approach for storing material parameters. The proprietary Synopsys Sentaurus device simulator uses a specific language for storing material data in files [17]. Overall, the discussed simulation tools use either an in-house language to store material data or use a hard-coded approach. Similar to COMSOL, our approach focuses on XML as the file format to store material data, offering improved support for data sharing among different applications. Furthermore, the simulation tools incorporate the material backend rather tightly into their software base, thus impeding reutilization of the material layer by other simulation tools. As our approach is based on a dedicated material library, we support different applications with a single implementation, fostering synergy effects.

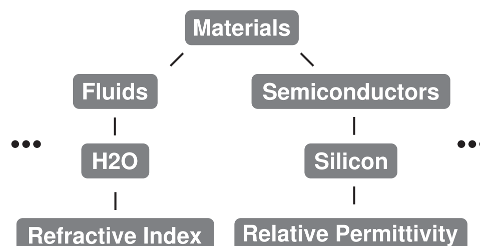



**Fig. 1.** Exemplary material properties schematically mapped on a tree. Materials do not necessarily share the same properties.

Several online databases are available, such as MatWeb [18], Springer Materials [19], and the Materials Project [20]. Aside of the obvious limiting factor of requiring an internet connection to access the data, different file formats used for potential exporting, and the restricted access via APIs, interfacing with simulation tools is impeded. Therefore, online resources can – if at all – be solely used to complement a material database and not act as the primary source for material parameters for a simulation tool. Our approach aims to fill this gap by supporting string-based queries as well as string-based attribute processing facilities, paving the way for remote network-based material databases providing data to local simulation tools.

This work introduces the free open source ViennaMaterials [21] library to fill the void of a reusable and dedicated materials platform for science and engineering applications. Our focus is on providing a C++-based library, offering APIs for C/C++ and Python applications; support for Fortran is planned. However, our software design supports extension towards other languages. These programming languages are the main languages used in computational science and engineering and were chosen for this reason. The goal of ViennaMaterials is to decouple applications from a single specific material database backend, thus increasing reusability and flexibility in the application development. Due to the mentioned advantages of using XML for storing material data, we concentrate on XML and XPath to store and query the material data, respectively.

The design as well as challenges and approaches are discussed in Section 2. Our work extends previous investigations in the area of handling material data via XML [22] by introducing software design issues and approaches as well as application examples. The favored XML layout for storing material data is introduced in Section 3. Two application scenarios are introduced to underline the feasibility of ViennaMaterials in Section 4: (1) ViennaMaterials is interfaced with a scientific simulation tool to provide the appropriate parameters for solving a Laplace problem based on a dependent material parameter. (2) A Python-based remote material database client–server setup is implemented, underlining the support for not only network-based applications, but also Python in general.

## 2. Software design

Fig. 2 depicts the general design concept of ViennaMaterials. Applications access the functionality of ViennaMaterials via APIs whereas the actual material data is stored in material files. The library design supports different APIs for various languages (cf. Section 2.1), such as C++ and Python. The APIs are available on top of a C++-based middleware (cf. Section 2.2), providing access to different storage backends as well as additional functionality: a language proxy module (cf. Section 2.3), allowing to simplify query expressions, a polymorphic material attribute module (cf. Section 2.4), supporting various types of material data (e.g. scalars and functions), a physical units module (cf. Section 2.5), enabling to process material data values associated with a physical unit, as well as a nested code evaluation module (cf. Section 2.6), essential for supporting mathematical material parameter models.

In the following sections, additional details of the respective aspects of ViennaMaterials are presented.

### 2.1. Application programming interfaces

As the primary programming language of ViennaMaterials is C++, the primary API is the C++ API. However, C++ is suited to be interfaced with other programming languages most important for science and engineering applications, those being C,

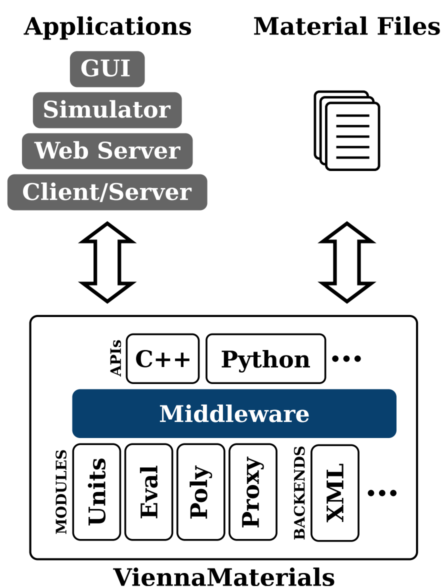



**Fig. 2.** The design of ViennaMaterials.

Fortran, and Python. C++'s built-in C-support is used to provide appropriate APIs for C and Fortran. We utilize SWIG [23] to provide a convenient access for Python-based applications.

### 2.2. Middleware

The middleware enables the applications to utilize modules and backends in a flexible and abstract manner (Fig. 3). According to the file formats of the input material files, the appropriate backend is selected. For instance, in case of XML, an XML/XPath-capable backend is instantiated and prepared for data access. We utilize the lightweight pugixml [9] library for the XML/XPath backend. Switching backends during run-time is realized via a C++ virtual class hierarchy, using dynamic polymorphism. The polymorphic methods enable the applications to query material data independent of the actual backend in use.

Upon receiving a query from the application via an API, the so-called *query processor* issues the appropriate queries (e.g. to extract the value and the unit of a parameter) to ultimately generate the desired result data to be returned to the application. The data retrieved from the material database is potentially represented by a mathematical expression, where variables denote dependencies. The middleware uses the nested code evaluation module (cf. Section 2.6) to evaluate the expression and to generate the actual parameter data. Furthermore, physical quantities are supported by coupling units with the material value to increase the robustness by ensuring that a value is linked to the correct unit (cf. Section 2.5). Material inheritance is not yet implemented but conceptually supported by our XML-based approach via retrieving the parent material from the child material, in turn triggering an internal subtree copy process from the parent material to the child material.

### 2.3. Language proxy

By default, the middleware expects all queries to be in the native query language of the backend, such as XPath. This fact potentially confines the application to the use of the backend language; although different backends can still be used, the backend had to support the application's provided query language for the data access to work. Consequently, switching to backends not supporting the query language used in the application, breaks the data access for the application.

Decoupling this approach in a generic manner is challenging, as it would require a query language translator mapping the input query language to the appropriate target language supported by the currently active backend. As we cannot foresee all possible query languages, we tackle this challenge by using an extendable language proxy mechanism. Each proxy provides a specific query path translation, allowing to decouple the applications from the actual backend languages. A dynamic polymorphism approach is used to define an abstract interface for the set of different proxies.

Fig. 4 schematically depicts the mechanism of a language proxy for XPath. An XPath-based path query is mapped to an arbitrary query language by using a proxy, allowing to perform automatic conversions of a foreign input query language to a defined target language assuming a specific XML structure (the context). However, the proxy expects a specific XML layout and a specific mapped language (the so-called *context*) to work, as the translation is basically based on substring substitutions. Consequently, if the XML layout changes, the proxy's translation mechanism must be updated, otherwise the target path does not fit the XML structure. However, this necessary adaptation is bearable as frequent changes in the XML

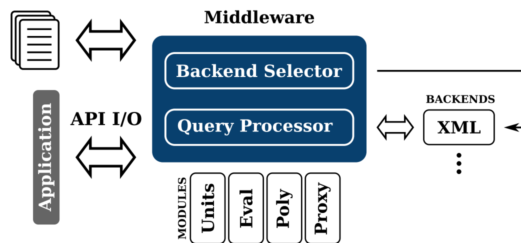



**Fig. 3.** The middleware orchestrates the backends and the modules.

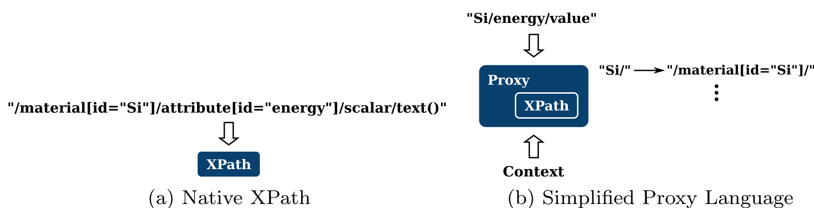



**Fig. 4.** The language proxy mechanism allows to perform automatic conversions of a foreign input query language to a defined target language.

layout are not a realistic scenario. Instead, it can be safely assumed that a language proxy is implemented once for a specific XML layout.

### 2.4. Polymorphic material data

Material data is multifaceted: the most basic form is a scalar-valued property. However, vectors or – more generally – tensors have to be supported as well. For instance, the electromagnetic permeability is represented by a scalar value in an isotropic medium, but a second-rank tensor is required for an anisotropic medium. Further contributing to the challenge of handling material data: a material property might depend on other properties, which is usually described by mathematical models. For instance, the electron mobility in semiconductor devices depends on temperature-dependent scattering processes.

Overall, the polymorphic nature of material properties introduces the need for distinguishing between the different types to allow the applications to adjust the handling of the material data objects appropriately. Such a dynamic type mechanism is common in scripting-oriented languages, such as Python. However, as our focus is on C++ – due to its excellent support for other programming languages as well as run-time performance by simultaneously offering a reasonable convenience layer – the goal is to provide such a dynamic type system for C++.

We approach this aspect by using a virtual class hierarchy, providing a generic interface to not only determine the kind of material property but also enabling access methods in a unified manner. More specifically, the interfaces allow to query the polymorphic data object for the type it contains. From this information, the actual data value (e.g. double) can be accessed by templated access methods, in turn using the type information to perform the appropriate type casts. Software developers can extend the set of supported data types by adding an additional specialization to the class hierarchy, overloading the virtual interface provided by the base class.

### 2.5. Physical units

A central aspect for increasing robustness of applications in science and engineering is the careful handling of physical units. This fact is especially true for material data, as the majority of data is associated with units, rendering a unit-agnostic mechanism essential. Further contributing to the challenge, the use of units not part of the international system of units is wide-spread, such as the centimeter–gram–second system frequently used in theoretical physics.

The common approach to incorporate unit handling in statically typed languages is to anchor it into the type system [24], meaning that a unit is associated with a unique type, allowing the compiler to raise errors at the earliest possible moment: during compilation. However, the task is to consider the inherent run-time environment we are focusing on with ViennaMaterials. This fact is not only imposed by the file-based material data, but also by applications utilizing ViennaMaterials, such as GUIs or client–server applications both favoring string-based data. Overall, as the unit data originates from files, the unit data is available in string-based objects. Therefore, the unit system must be capable of processing string-based unit expressions, requiring significant parsing and evaluation facilities to incorporate a dynamic unit system.

We approach these issues by using a C++ quantity class holding a value and a string-based unit object. Although this approach already allows to link the value with a unit, it is not really unit-agnostic. For instance, a proper unit system identifies Watt (`"W"`) to be the same unit as Voltampere (`"VA"`), a direct string-based comparison will fail. Therefore, we couple the quantity class with the UDUNITS [25] library, a C-based library supporting string-based unit expressions and conversions. ViennaMaterials' middleware interfaces with the quantity class as well as with the UDUNITS library to provide the required unit-agnostic material data experience to the applications and to the models stored in the database files.

### 2.6. Nested code evaluation

As already indicated, an essential part of material properties are mathematical models, allowing to describe intricate material properties by analytic functions. These mathematical models potentially require at least one input parameter, typically only known to the application requesting the material property. For instance, a temperature distribution stored on the mesh elements within a simulation tool could be used to evaluate a mathematical material model on each mesh element.

This fact puts pressure on simplistic material parameter database approaches as it requires functionality beyond straightforward data access. In fact, evaluating mathematical material models requires to process the associated mathematical expression - stored in the material database – upon the application's data access. Another issue is the run-time setting, i.e., the knowledge of the structure and the set of dependent input variables of the expression are only available during run-time. This requires the code evaluation module to enable external applications access to set the input parameters via the APIs.

We tackle this challenge by providing a nested code evaluation module, where code can be implemented within material parameters stored in the database files (cf. Section 3). In turn, the presence of a code representing a material parameter is detected by ViennaMaterials' middleware, and the code evaluation module resolves the mathematical expression contained in the string-based code retrieved from the material database file. To uphold flexibility, our code evaluation backend is based on C++ dynamic polymorphism techniques, allowing to switch code evaluation backends during run-time. This allows ViennaMaterials' middleware to instantiate the appropriate code execution backend based on string-based information

indicating the type of the code's programming language. Such a string-based code language indicator can be easily provided by the input material database files as part of the stored material parameter data. The support for nested code expressions also fosters the incorporation of external models stored in external files, as available implementations can be loaded and accessed from within the nested code. Our primary code evaluation backend is Python due to its flexibility, rich ecosystem, and widespread application, fostering model re-utilization within ViennaMaterials. To this end, we use the Python C-API, supporting the execution of string-based Python code, including support for input arguments and return values (Fig. 5). The code evaluation module receives a string-based code language identifier (1), the actual code (2), and the input parameters (3) from the middleware and application, respectively. According to the code language identifier, the appropriate backend is activated (e.g. Python) via a language dispatch mechanism, the input arguments are set, the code is executed, and the result is returned via the code evaluation routine. Although we are currently focusing on Python as the primary *nested* language, our approach can be naturally extended to support compiler-based languages via a just-in-time compilation approach.

The input parameters associated with the mathematical expressions can be queried by the external applications via the polymorphic object data (cf. Section 2.4). The name of each input parameter as well as the ability to set the respective quantity is provided.

A particular difficulty is the synchronization between the parameter names required by a model and the names available in an application. Even if the same quantity is referred to, simple string comparison used by an application to identify the required quantity fails to correctly map, for instance, `Temperature` to `temp`. Such a problem arises when a foreign material input file is used which uses different parameter names as an application. It is thus required to create an associative quantity name container within the application to provide a correct mapping. This is necessary, as ViennaMaterials cannot foresee all possible quantity names and notations.

Furthermore, the code evaluation mechanism enables to support more intricate data structures. For instance, material data may be available in the form of tables, each entry holding measured values for dependencies between the particular quantities where the intermediate values are approximated linearly. In such cases, the data tables could be implemented in the code, potentially utilizing convenience libraries such as NumPy [26]. Additionally, data tables available in external files can be accessed via the appropriate nested code expression for importing file-based data. These facts further underline the advantage of using Python with its vast ecosystem.

## 3. XML material data layout

In this section we discuss our XML layout used to store material data. In particular we cover the challenges of storing units as well as the different ways of representing material data, e.g., scalar-valued or mathematical model-based parameters.

A material is stored in the corresponding `material` XML element, in turn allowing to assign an additional XML element, such as identifier (`id`), full name (`name`), categories (`category`), notes (`note`), and attributes (`attribute`) or groups (`group`), which are discussed in the following.

```
<material>
  <id>Si</id>
  <name>silicon</name>
  <category>semiconductor</category>
  <note>Silicon</note>
  .. attributes or groups ..
</material>
```

Each material property is stored in an `attribute` XML element, which additionally is build upon an `id` XML element, an XML element holding the actual property data, such as scalar values (`scalar`), and a `unit` XML element. In the following, an exemplary scalar-valued material attribute is shown.

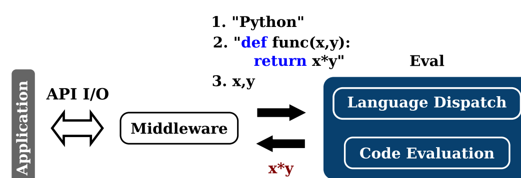



**Fig. 5.** The evaluation module handles the execution of nested code.

```
<attribute>
  <id>eps_r</id>
  <name>relative permittivity</name>
  <scalar type='float'>11.6</scalar>
  <unit>1</unit>
</attribute>
```

As can be seen from this example, the `scalar` XML element supports different XML attributes indicating the data type of the stored material property value. This allows ViennaMaterials' middleware to use the appropriate conversion mechanism for exposing the actual data object via the APIs to the applications (cf. Section 2.4). Currently `boolean`, `integer`, and `float` types are supported, being converted to `bool`, `long`, and `double` C++ data types, respectively.

For storing tensors, we use a dedicated `tensor` XML element to be used inside the `attribute` XML element, as is shown in the following for an exemplary 2-tensor:

```
<tensor order=2 d1=2 d2=2 type='float'>
<scalar i0=0 i1=1>2.0</scalar>
</tensor>
```

The tensor `order` as well as the individual dimensions (e.g. `d1`) are stored. We use a sparse storage format, meaning that only tensor entries unequal zero are explicitly stored via the appropriate indices (e.g. `i0`), reducing the space requirements considerably. The above storage format scales for arbitrary tensor orders and dimensions.

Mathematical material property models are stored using the `function` XML element, allowing to hold arbitrary code expressions as well as input and output parameters, as is shown in the following.

```
<function>
  <code lang='python' call='add'>
    def add(x, y):
    return x+y+1000.0
  </code>
  <arg>
    <id>0</id>
    <name>x-factor</name>
    <scalar type='float'>2.0</scalar>
  </arg>
  <arg>
    <id>1</id>
    <reference>/*/*[id='material-name']/*[id='parameter']
    </reference>
  </arg>
  <return>
    <scalar type='float'/>
  </return>
</function>
```

The `code` XML element contains the code expression, in this case a Python function (Lines 2–5). By providing the `lang` and `call` XML attributes, the middleware can select the appropriate code evaluation backend (in this case Python); in turn, the Python code evaluation backend can load and execute the function (cf. Section 2.6). For the latter, the Python library requires to provide the name of the method to be executed. By using the `arg` elements an arbitrary number of input parameters can be defined, implicitly assuming that the `id` value corresponds to the position in the function's argument list (Lines 6–10). A default value for the argument can be provided (Line 9). The name of the dependency is stored in the `name` element, allowing the application to access it. This enables the application to perform a mapping from an internally used quantity name to the one stored as a dependency in the material database. References to other material parameters are supported

by nesting a `reference` element into an `arg` element (Lines 11–15). Internally, an access to such a quantity reference triggers an access redirection in form of an additional query based on the stored path. The `return` element enables the middleware to prepare the appropriate result type based on the polymorphic material data (cf. Section 2.4).

Aside from the `category` association element, the materials or attributes can be grouped together by the `group` element, providing an additional and more intuitive way of organizing material data.

```
<group>
  <id>some group name</id>
  .. materials or attributes ..
</group>
```

As already indicated, applications utilizing ViennaMaterials can use the API to access the database. For instance, the XPath query language can be used to request specific data sets via its native filter mechanisms. Based on the above XML structure, Table 1 depicts exemplary XPath queries to generate a list of all material names, attributes of silicon, models of silicon, and all model dependencies of silicon. The return objects are string-based and can thus be directly used to populate GUIs or to generate more refined queries.

## 4. Applications

This section depicts two use cases of ViennaMaterials. Section 4.1 discusses interfacing ViennaMaterials with a typical simulation based on partial differential equations. Section 4.2 investigates utilization within a Python-based client–server application enabling to setup a remote material database.

### 4.1. Scientific simulation

A Laplace problem is solved for a solution variable $u$ (a potential or a stationary density of a diffusing material) for a one-dimensional multi-layered device, based on a coefficient $\eta(T)$, which depends linearly on the temperature $T$

$$\eta(T)\nabla^2 u = 0.$$

Such a problem statement represents the homogeneous and stationary diffusion equation. The coefficient $\eta(T)$ represents a material-dependent parameter which is loaded from a ViennaMaterials database. Constant material quantities are stored as scalar-valued data, whereas temperature-dependent parameters are stored as mathematical models, defining the temperature as a dependency.

Interfacing a simulator with ViennaMaterials requires the simulator to issue a query regarding the material-dependent coefficient $\eta(T)$. The returned polymorphic material data object is used to identify the actual type it holds; the API allows to determine whether the parameter is in fact a scalar-, a tensor-, or a function-based quantity. If indeed the returned material parameter is a function, the name and type of the required input parameters are extracted, set, and ultimately the function is executed via the nested code evaluation module (cf. Section 2.6) ultimately yielding the result quantity.

In a simulator setting, the information of the material type is inherently available as run-time data originating, for instance, from GUIs. This fact typically yields string-based material names chosen by an end-user for each individual material of the device under consideration. Therefore, the simulator needs to process the string-based material names and requests the appropriate data from ViennaMaterials. For instance, a GUI application provides the ability to select a material from a list of materials for a certain region of the simulation domain. This string is then mapped to the corresponding ViennaMaterials material identifiers as are the required material parameters by the simulator. The mapped keys are used by the simulator to perform the necessary queries.

The device under consideration in this example consists of five segments (i.e. layers); two metal contact segments, both set to Dirichlet boundary conditions, are attached to either side of a three-layered structure, where the middle layer offers a time-dependent coefficient and the surrounding two layers offer a constant coefficient. In this particular case, the material parameters are requested by the simulator for each material layer, therefore in this case three queries are conducted. For performance reasons, material data queries should not be conducted within performance sensible code regions, such as

**Table 1**
Exemplary XPath queries.

| | |
|---|---|
| Materials | `/material/id` |
| Attributes | `/material[id="Si"]/attribute/id` |
| Models | `/material[id="Si"]/attribute[descendant::function]/id` |
| Dependencies | `/material[id="Si"]/attribute/function/arg/name` |

compute-intensive loops. The material properties are then distributed over the respective simulation domain by the simulator, the partial differential equation is discretized, the equation system is assembled using a finite volume method, which in turn is being solved by a linear solver. The normalized solution of a Laplace problem for a temperature- and a material-dependent coefficient parameter is compared on top of a one-dimensional device with a length of 4 m (Fig. 6). Different point types and colors denote different temperatures, i.e., 223 K, 273 K, 323 K. Dirichlet boundary conditions (DC) are assigned to the right ($x \geqslant 3$) and to the left layer ($x < 0$). The middle layer ($1 \leqslant x \leq 2$) offers a temperature-dependent coefficient $\eta(T)$, whereas the surrounding layers ($0 \leqslant x \leq 1$ and $2 \leqslant x \leq 3$) have a constant coefficient $\eta$.

### 4.2. Remote material database

This application example depicts the utilization of ViennaMaterials to provide a remote material database by using Python as is shown in Fig. 7. The server application, interfaced with ViennaMaterials, prepares for incoming connections requesting material data. The client application uses a network connection to request data from the server.

The use case of requesting a mathematical material model by a client application from the server is shown, followed by a client-side generation of a parameter curve by evaluating the model for a range of input values. More concretely, the lattice mobility model for electrons is evaluated for silicon and germanium [3]. The model is stored via the `function` tag (cf. Section 3) and processed by the nested code evaluation module (cf. Section 2.6). This application underlines the support for various use cases covering not only simulation tasks but also more exotic tasks, such as a remote web-server based material database with material parameter visualization capabilities.

Concerning the network communication, Python's *socket* module is used, allowing direct string-based network transfer by using target/source internet protocol addresses and ports.

A simplified implementation of the Python-based server application is as follows.

```
materialdb = pyviennamaterials.library(matfile)
mySocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mySocket.bind((host, port))
mySocket.listen(1)
while True:
connection,client_address = mySocket.accept()
query = connection.recv(buffersize)
connection.sendall(materialdb.query_to_string(query))
connection.close()
```

The ViennaMaterials library is generated by loading the appropriate input material database file (Line 1). A Python socket is opened on the host with a specific port (Lines 2–4). Upon an incoming connection, the data (i.e. the database query) is received up to a maximum length of the data stream indicated by `buffersize` (Lines 6–7). The query is forwarded to the ViennaMaterials library, and the result string is returned to the client (Lines 8). The connection is closed (Line 9) and the socket waits for a new incoming connection (Line 5). Note that the data transmitted over the network only supports string-based objects. This requires the use of a specialized query, yielding raw string-based XML data which can be natively transmitted over the network.

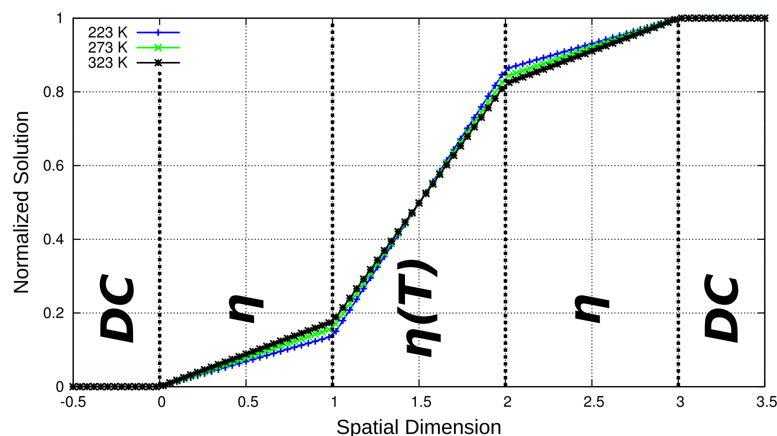



**Fig. 6.** The normalized solution of a Laplace problem for a temperature- and a material-dependent coefficient parameter on top of a five-layered one-dimensional structure. DC refers to Dirichlet boundary condition and $\eta, \eta(T)$ to constant and temperature-dependent coefficients, respectively.

A suitable and simplified stand-alone Python client application is as follows.

```
mySocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mySocket.connect((host,port))
mySocket.sendall(query)
result = mySocket.recv(buffersize)
attribute = pyviennamaterials.generate_attribute(result)
func_args = attribute.get_dependencies()
data_table = list()
for T in range(100,510,10):
  func_args[0].set_value(float(T))
  attribute.set_dependencies(func_args)
  data_table.append([T, attribute.evaluate_value_float()])
```

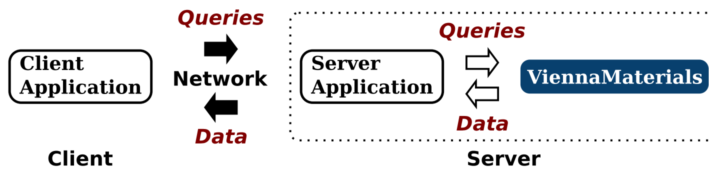



**Fig. 7.** A schematic overview of a client–server-based material database setup.

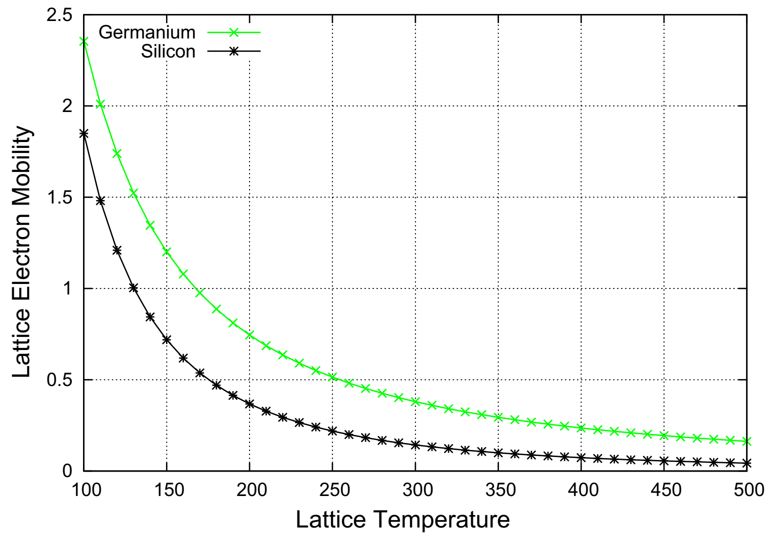



**Fig. 8.** Comparison of the lattice mobility (in $m^2/Vs$) for electrons between silicon and germanium and for a temperature range of 100–500 K.

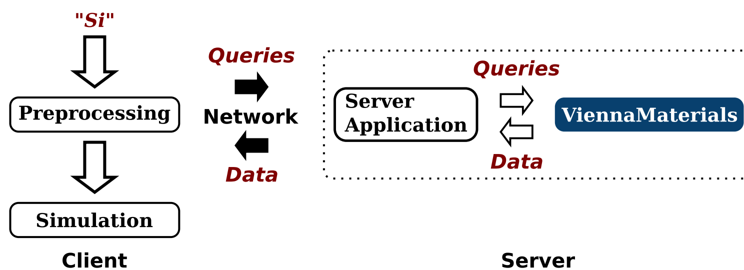



**Fig. 9.** A simulation tool can be coupled with a remote material database.

A new socket connection is opened to the host using a specific port (Lines 1–2). The query is transmitted (Line 3) and the string-based result data is received (Line 4). Auxiliary quantity generation and evaluation methods provided by ViennaMaterials (not requiring access to the actual database) are utilized to process the result string and to generate a polymorphic material data object (Line 5). This polymorphic material data object is used to retrieve the input dependencies (Line 6). A range of temperature values is generated and based on these values the material model is evaluated, yielding a result table mapping temperature values to mobility values. Fig. 8 depicts the chart visualizing the lattice mobility values for electrons and for silicon and germanium.

The client–server approach discussed here can be implemented on top of the previously introduced simulation setup (Section 4.1) as is sketched in Fig. 9. Such a setup would enable simulation tools to switch between material databases provided by, for instance, collaborators, thereby having access to potentially updated data without exchanging files or recompiling the simulator. This fact underlines the significant advantage of decoupling the material backend from the simulation tools by using ViennaMaterials.

## 5. Summary

The dedicated material library ViennaMaterials has been introduced. The motivation, challenges, and approaches in the form of software design aspects have been discussed as well as application examples have been shown. Our primary research contribution is the support for code blocks based on different programming languages representing mathematical material models. This technique augments a plain XML-based material data approach with support for mathematical models. Furthermore, our language proxy approach allows to decouple applications from backend languages, further increasing usability. Due to our focus on the run-time regime, applications with graphical user interfaces or network-based remote applications are natively supported. Especially the latter has potential for changing the way scientists and engineers access data for their simulation tools: by switching material database servers, quick access to remote databases feeding material data to the local simulation is accomplished. This gives rise to the possibility for researchers to operate their own material database servers, and provide material data for educational or professional purposes. Our approach shows that XML and XPath are excellent instruments for implementing a material library. Future work will focus on the performance aspects of the nested code evaluation mechanism as well as on supporting complex-valued material parameters, material inheritance, and providing a ready-to-use XML-based material file.

## Acknowledgment

This work has been supported by the Austrian Science Fund (FWF) through the Grants P23296 and P23598, and the European Research Council (ERC) through the Grant #247056 MOSILSPIN.

## References

[1] M. Gayer, G. Iannaccone, A software platform for nanoscale device simulation and visualization, in: Proceedings of the International Conference on Advances in Computational Tools for Engineering Applications (ACTEA), 2009, pp. 432–437. doi:http://dx.doi.org/10.1109/ACTEA.2009.5227880.
[2] A. Logg, G.N. Wells, DOLFIN: automated finite element computing, ACM Trans. Math. Softw. 37 (2) (2010) 1–28, http://dx.doi.org/10.1145/1731022.1731030.
[3] S. Selberherr, Analysis and Simulation of Semiconductor Devices, Springer, 1984, ISBN 3211818006.
[4] B. Stroustrup, Software development for infrastructure, Computer 45 (1) (2012) 47–58, http://dx.doi.org/10.1109/MC.2011.353.
[5] W.E. Brown, Introduction to the SI library of unit-based computation, in: Proceedings of the International Conference on Computing in High Energy Physics (CHEP), 1998.
[6] W.E. Brown, Applied template metaprogramming in SIUNITS: the library of unit-based computation, in: Proceedings of the Workshop on C++ Template Programming, 2001.
[7] M. Kenniston, Dimension checking of physical quantities, in: C/C++ Users Journal, 2002.
[8] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming, Addison-Wesley, 2004, ISBN 0321227255.
[9] Pugixml. <http://pugixml.org/>.
[10] Libxml2. <http://www.xmlsoft.org/>.
[11] R. Klima, Three-dimensional device simulation with minimos-NT (Dissertation), Technische Universität Wien, 2002. <http://www.iue.tuwien.ac.at/phd/klima/>.
[12] ViennaIPD. <http://viennaipd.sourceforge.net/>.
[13] COMSOL multiphysics. <http://www.comsol.com/>.
[14] H.G. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, Comput. Phys. 12 (6) (1998) 620–631, http://dx.doi.org/10.1063/1.168744.
[15] K. Rupp, T. Grasser, A. Jüngel, On the feasibility of spherical harmonics expansions of the Boltzmann transport equation for three-dimensional device geometries, in: Proceedings of the IEEE International Electron Devices Meeting (IEDM), 2011, pp. 34.1.1–34.1.4. doi:http://dx.doi.org/10.1109/IEDM.2011.6131667.
[16] J. Sellier, J. Fonseca, G. Klimeck, Archimedes, the free Monte Carlo simulator, in: Proceedings of the International Workshop on Computational Electronics (IWCE), 2012, pp. 1–4. doi:http://dx.doi.org/10.1109/IWCE.2012.6242861.
[17] Synopsys Sentaurus. <http://www.synopsys.com/tools/tcad/>.
[18] MatWeb. <http://www.matweb.com/>.
[19] SpringerMaterials. <http://www.springermaterials.com/>.
[20] The Materials Project. <https://materialsproject.org/>.
[21] ViennaMaterials. <http://viennamaterials.sourceforge.net/>.
[22] J. Weinbub, R. Heinzl, P. Schwaha, F. Stimpfl, S. Selberherr, A lightweight material library for scientific computing in C++, in: Proceedings of the European Simulation and Modelling Conference (ESM), 2010, pp. 454–458.

[23] SWIG. <http://www.swig.org/>.
[24] Boost units. <http://www.boost.org/libs/units/>.
[25] UDUNITS. <http://www.unidata.ucar.edu/software/udunits/>.
[26] NumPy. <http://www.numpy.org/>.