# A Lightweight Material Library for Scientific Computing in C++

Josef Weinbub, René Heinzl,
Franz Stimpfl, Siegfried Selberherr

Institute for Microelectronics, TU Wien
Gußhausstraße 27-29 / E360
1040 Vienna, Austria
{weinbub|heinzl|stimpfl|selberherr}@iue.tuwien.ac.at

Philipp Schwaha

Shenteq s.r.o.
Záhradnícka 7
81107 Bratislava, Slovak Republic
schwaha@shenteq.com

## ABSTRACT

Simulations in the field of scientific computing require often the availability of large sets of material properties. We propose a convenient approach for a lightweight material library using available open source tools. The presented approach is therefore suited for embedding into larger projects, such as simulators. The XML file format as well as an XML parser library is used to store, load, and manage the data. The location of data items or data sets is specified using XPath query language. Furthermore, an utility is provided for the conversion of the initially untyped data items to the numerical data types required by the simulation package. As performance is an issue in this context, we present a simple use case.

## INTRODUCTION

Simulation tools require often a large set of (material) parameters to carry out scientific simulations [1, 2], due to the use of equations which include material parameters to model the physical environment. Among such equations, partial differential equations are especially wide spread in the description of complex phenomena and are therefore of special interest for scientific computing. A very prominent example is the system of Maxwell equations [3]:

$$\vec{\nabla} \times \vec{E} = -\partial_t \vec{B} \tag{1}$$

$$\vec{\nabla} \cdot \vec{B} = 0 \tag{2}$$

$$\vec{\nabla} \times \vec{H} = \vec{J} + \partial_t \vec{D} \tag{3}$$

$$\vec{\nabla} \cdot \vec{D} = \rho \tag{4}$$

These four equations can be split into two almost independent pairs. The first pair consists of Equation 1 and Equation 2, which relates the spatial and time derivatives of the vector fields $\vec{E}$ and $\vec{B}$ representing the electrical field strength and the magnetic flux density, respectively.

The second pair is based on Equation 3 and Equation 4, which links the spatial and time derivatives of the the magnetic field strength $\vec{H}$ and the electrical flux density $\vec{D}$. Note, that $\vec{J}$ denotes the vector field of the current density and $\rho$ the charge density.
The Maxwell equations themselves set up a formal structure in each of the pairs, which links magnetic and electric field components. However, only by combining both sets of equations a complete dynamical system which carries energy and momentum can be achieved. This attachment is accomplished using the material relations (Equations 5a, 5b), which emphasizes the important role of material properties:

$$\vec{D} = \varepsilon \vec{E} \tag{5a}$$

$$\vec{B} = \mu \vec{H} \tag{5b}$$

These equations are of special interest, as they relate flux densities to field strengths by material properties. Here $\varepsilon$ denotes the permittivity, and $\mu$ the permeability. Both relations appear very simple, but both permittivity and permeability may need modeling using complex, nonlinear functions depending not only on the material, but also on the magnitude of the encountered field quantities.
Considering the vast number of phenomena and the related sets of equations for which simulation environments have been and are currently being 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 and fast data access. Another important design goal is to embed the material library into simulator environments in an orthogonal fashion, as is schematically depicted in Figure 1.
This conceptual design not only results in basic modularity during software development, but also entails that the individual modules can be changed freely without affecting the other modules. This design therefore facilitates extendability and maintainability. While flexibility is a major goal, it must not compromise the consumption of computing resources.

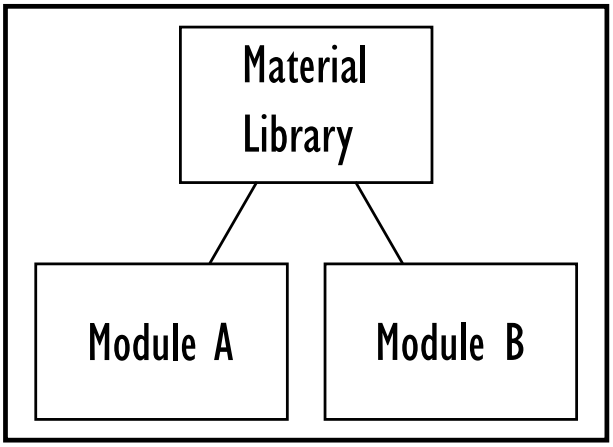


**Figure 1:** The material library can be part of a module set. Combining different modules results in a full scale application.

Therefore, the implementation of the material library should be as lightweight as possible. As such, memory consumption and the size of the implementation are also considered, when measuring the lightweight nature of the developed library, besides the run time performance. The modular nature ensures, that the use of the material library does not compromise the application which it is part of logically, while the lightweight nature ensures, that resources are not squandered needlessly.
At last, the run time performance of access to data is especially important, as repeated data access is typical during simulations. Therefore, minimizing the access times is important to maintain simulation performance.

## LIBRARY STRUCTURE

XML [4] has been chosen as the storage format for the material data (Figure 2). The underlying data associated with materials is inherently hierarchical, as shown in Figure 3, and can be mapped to a tree naturally, which is stored using XML. The variation due to the fact that not all parameters are available or useful for all materials can also easily be accommodated by the flexible nature of XML.

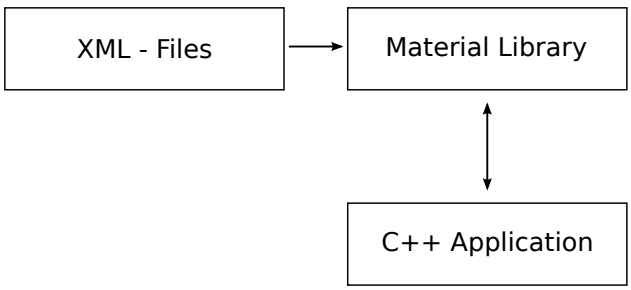


**Figure 2:** XML files are used as input for the material library. Data can be accessed by C++ applications.

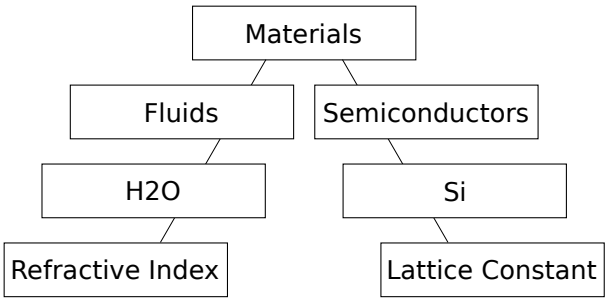


**Figure 3:** Material Properties schematically mapped on a Tree. Materials do not necessarily share the same properties.

The provided examples and use cases are related to embedding the XML parser library in a simulator from the field of device simulation. Since the goal is to provide a lightweight module with an already existing framework, existing client/server database approaches have not been used, as for example with PostgreSQL [5].

The material library is implemented based on a lightweight XML parser library named PugiXML [6], which is implemented in C++. Due to the basic functionality of the parser library, it is well suited to be embedded in a C++ framework. In addition to the bare XML parsing facilities, PugiXML also offers an implementation of the XPath 1.0 [7, 8] querying language. Note, that other parsing libraries have been investigated as well. Prominent examples include Xerces-C [9] and XQilla [10]. Although, XQilla offers support for XPath 2.0, it is based on the rather old and monolithically implemented Xerces-C parser. It has therefore been discarded, as it conflicts with the goals of a lightweight C++ implementation.
RapidXML has also been dismissed, as it does not offer any support for a querying language for data access. On the other hand, TinyXML [11] allows for support for a querying language to be added by a separately implemented XPath 1.0 library, named TinyXPath [12]. However, due to the fact, that PugiXML natively provides XPath support, it is the XML library of choice. It should, however, not go unnoticed that should the need arise, the XML library back end can easily be exchanged, as long as it provides XPath facilities.

## XML FILE SETUP

This section discusses the chosen file setup used for the XML input files. The following XML snippet depicts parts of the schematic mapping introduced in Figure 3 thus yielding a hierarchical database.

```
<db>
  <ele>
    <id>Materials</id>
    <ele>
      <id>Semiconductors</id>
      <ele>
        <id>Si</id>
      </ele>
    </ele>
  </ele>
</db>
```

Each element has the following general setup:

```
<ele>                 // introduce a new element
  <id>name</id>       // identifiy this element
</ele>
```

Note, that the tag names do not change. The name specified within the *id* tag is used for identification. In order to uniquely accommodate several materials with the same name, unique ids, for example assigned consecutively, may be introduced.

The actual data, only text in this case, is stored in the value fields of the nodes corresponding to the various tags. This approach enables to setup hierarchies of arbitrary depth. Hence, imposing no restrictions on the setup of the database.

To store actual data values, a special node hierarchy has been chosen to not only support the commonly used floating point numbers, but also different representations, which may be more suitable under certain circumstances. The following XML snippet depicts a property node with data and representation child nodes.

```
<props>
  <data>
    <id>Lattice Constant</id>
    <repr>
      <double>0.543072</double>
    </repr>
    <unit>nm</unit>
  </data>
  <data>
    <id>Dielectric Constant</id>
    <repr>
      <double>11.8</double>
      <rational>59/5</rational>
    </repr>
  </data>
</props>
```

Note, that for each new property a new data node, with a related identifier, is introduced. Different representations can be embedded within the data node. This approach is especially of interest for robust applications, as the most suitable representation of a data value can be chosen at execution time. Additional representations can be added as the need arises. Similarly to the different representations, different units can also be specified in distinct nodes.

## QUERY

The access to data is implemented by using the XPath query language. The use of a query language greatly enhances the flexibility of data access. There is no need for an additional data access layer via the Application Programming Interface (API), as the query language allows direct access to the data.
In the following, several queries are discussed to illustrate the functionality. These queries make use of the following XML data structure:

```
<db>
  <ele>
    <id>Materials</id>
    <ele>
      <id>Semiconductors</id>
      <ele>
        <id>Si</id>
        <props>
          <data>
            <id>Lattice Constant</id>
            <repr>
              <double>0.543072</double>
            </repr>
            <unit>nm</unit>
          </data>
          <data>
            <id>Dielectric Constant</id>
            <repr>
              <double>11.8</double>
              <rational>59/5</rational>
            </repr>
          </data>
        </props>
      </ele>
    </ele>
  </ele>
</db>
```

Common queries return the subtree of the data structure with the root node derived from the lowest query element. The following query accesses a specific property, for example.

```
db/ele/ele/ele/props/data[id="Lattice Constant"]
```

The XPath syntax syntax facilitates the intuitive descent along the structure of the tree holding the data. The values of the selected subtrees can then be investigated as the need arises.
As a result, the following subtree is returned.

```
<data>
  <id>Lattice Constant</id>
  <repr>
    <double>0.543072</double>
  </repr>
  <unit>nm</unit>
</data>
```

Note, that the whole node is again returned in XML format. This enables further processing of the returned data using the same mechanisms. It enables to conveniently investigate the database. This approach can be used to quickly partition and browse large databases.

However, in the case of a material database for simulations, the most common task is to directly access the data values. To directly access the values, the *text()* node can be used for an arbitrary node, as is shown in the following query

```
db/ele/ele/ele/props/data[id="Lattice Constant"]
/repr/double/text()
```

which results in:

```
0.543072
```

## CONVERSION

For numerical applications the retrieved string values must be converted to numerical data types, for example, *double*. A polymorphic data type with a run time evaluation system based on the Boost Spirit Parser facilities [13] is used to this end.
At the moment, the conversion utility is only capable of dealing with double values. If the value being parsed can not be handled as a double, it is kept as a string. However, the parser facility can be conveniently extended to support additional conversion targets, such as, integer values or rational numbers for example.
The following C++ code snippet depicts the behaviour.

```
// parse the value result, get poly result
poly_data pres = convert(value_string);

// test poly result on a certain data type
// extract the data accordingly
if( is< numeric_type >(pres) ) {
   numeric_type dblval=get<numeric_type>(pres);
}
if( is< string_type >(pres) ) {
   string_type strval=get<string_type>(pres);
}
```

## PERFORMANCE

This section introduces a few performance statistics for different input XML files. The query execution performance, the peak memory usage and the time required to load a file is investigated. The test platform is a PC with an AMD Phenom II X4 - 965 CPU and 8GB of RAM. The operating system is a Funtoo Linux [14] 64-bit with a 2.6.34 kernel.
Table 1 depicts the query execution performance for input files of different sizes. To investigate high load, 1E6 queries have been executed. Note, that the similar query execution times of the larger files is due to the fact, that the query depth is equally long, which is 9 for those files. Whereas, the query depth of the smaller files, is 3 levels smaller. Apparently, the query depth influences the performance more significantly, than the file size. To improve the execution performance of queries, especially for large files, the XML hierarchy should be as flat as possible, so the query depths are kept small.

| *XML File Size* | *Total Time* | *Time per Query* |
|---|---|---|
| 1.5 KB | 5.66 $s$ | 5.66 $\mu s$ |
| 42.2 KB | 18.15 $s$ | 18.15 $\mu s$ |
| 3.1 MB | 121.63 $s$ | 121.6 $\mu s$ |
| 6.2 MB | 126.86 $s$ | 126.8 $\mu s$ |
| 9.7 MB | 132.54 $s$ | 132.5 $\mu s$ |

**Table 1:** Overview of query execution performance for 1E6 queries based on input XML files of different sizes.

| *XML File Size* | *Peak Memory* | *File Loading* |
|---|---|---|
| 1.5 KB | 38.17 KB | <1ms |
| 42.2 KB | 181.1 KB | <1ms |
| 3.1 MB | 10.66 MB | 13ms |
| 6.2 MB | 21.26 MB | 26ms |
| 9.7 MB | 25.40 MB | 32ms |

**Table 2:** Overview of the peak memory consumption and the file loading performance based on input XML files of different sizes.

Table 2 depicts the peak memory usage, which has been measured with Valgrind [15], and the file loading time. Note, that the peak memory consumption can be considered exceptional as for a input file of roughly 10 MB the maximum amount of required memory is only around 25 MB. This fact emphasizes the applicability of this approach as a lightweight database for applications in the field of scientific computing.
Furthermore, the implementation of PugiXML is based on only four source files, which have roughly 280 KB of total size. Therefore, it can be easily added to a project as an external, third-party library.

## USE CASE EXAMPLE

This section depicts a usecase to illustrate the application of the introduced approach in a C++ environment. The goal is to setup the library, load input XML data, and access the data by the query language. Finally, the result of the query is converted to a numerical datatype.

```
// the datastructure is instantiated
pugi::xml_document doc;

// the xml file is loaded
pugi::xml_parse_result result =
   doc.load_file("input/dev.xml");

// a query string is set up
std::string query_string("db/ele/ele/ele/props
   /data[id=\"Lattice Constant\"]
   /repr/double/text()");

// the query string is processed ..
pugi::xpath_query query(query_string.c_str());

// .. and evaluated
pugi::xpath_node_set tools =
   query.evaluate_node_set(doc);
```

Finally the string typed value has to be converted into a numerical datatype.

```
// the result is converted to a string
std::stringstream resultstream;
tools[0].node().print(resultstream, "  ");
std::string value_string(resultstream.str());

// parse the value result, get poly result
poly_data pres = convert(value_string);

// the result is tested if it is a numeric
// type and the data is extracted accordingly
if( is< numeric_type >(pres) ) {
   numeric_type quan=get<numeric_type>(pres);
```

## CONCLUSION

A fast and lightweight application of a XML parser library as a material library has been introduced. The XML file setup has been discussed as well as the usage of the query language to access the data. A conversion utility enables the use of this approach in numerical applications. The peak memory consumption as well as the file loading times and the query execution times have been investigated for input files of different sizes. A use case example depicts the implementation details for using the presented approach, as well as a possible application scenario as part of a simulator environment is introduced.

## ACKNOWLEDGMENTS

The authors want to thank Karl Rupp from the Christian Doppler Laboratory for Reliability Issues in Microelectronics from TU Wien for his valuable input. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN and by the Austrian Science Fund FWF, project P19532-N13.

## REFERENCES

[1] M. Gayer and G. Iannaccone, "A Software Platform for Nanoscale Device Simulation and Visualization," in *Advances in Computational Tools for Engineering Applications, ACTEA*, 15-17 2009, pp. 432–437.

[2] A. Logg and G. N. Wells, "DOLFIN: Automated Finite Element Computing," *ACM Transactions on Mathematical Software*, vol. 37, no. 2, pp. 1–28, 2010.

[3] J. C. Maxwell, *A Treatise on Electricity & Magnetism*. New York: Dover Publications, 1873.

[4] *Extensible Markup Language (XML) 1.0*, http://www.w3.org/TR/REC-xml.

[5] *PostgreSQL*, http://www.postgresql.org.

[6] *PugiXML*, http://code.google.com/p/pugixml.

[7] *XML Path Language (XPath) 1.0*, http://www.w3.org/TR/xpath.

[8] M. Benedikt *et al.*, "XPath Leashed," in *In ACM Computing Surveys*, 2007.

[9] *Xerces-C++ Parser*, The Apache Software Foundation, http://xerces.apache.org.

[10] *XQilla*, http://xqilla.sourceforge.net.

[11] *TinyXml*, http://sourceforge.net/projects/tinyxml.

[12] *TinyXPath*, http://tinyxpath.sourceforge.net.

[13] J. Weinbub *et al.*, "A Dispatched Covariant Type System for Numerical Applications in C++," in *International Conference of Numerical Analysis and Applied Mathematics, ICNAAM*. AIP Conference Proceedings, 2010, accepted.

[14] *Funtoo Linux*, http://www.funtoo.org.

[15] *Valgrind*, http://valgrind.org.

## BIOGRAPHY

**JOSEF WEINBUB** studied electrical engineering and microelectronics at the *Technische Universität Wien*, where he received the degree of *Diplomingenieur* in 2009. He is currently working on his doctoral degree, where his scientific interests are in the field of scientific computing, with a special focus on algorithms and datastructures, modern programming techniques, and high-performance computing.

**RENÉ HEINZL** studied electrical engineering at the *Technische Universität Wien*, where he received the degree of *Diplomingenieur* in 2003 and his PhD in technical sciences in 2007. His research interests include programming paradigms, high performance programming techniques, data structural aspects of scientific computing, performance analysis, process simulation, solid modeling, scientific visualization, algebraic topology, and mesh generation and adaptation for TCAD.

**FRANZ STIMPFL** studied computer science at the *Technische Universität Wien*, where he received the degree of *Diplomingenieur* in 2007. He joined the Institute for Microelectronics in October 2007, where he is currently working on his doctoral degree. His research activities include mesh generation and modern software paradigms.

**PHILIPP SCHWAHA** studied electrical engineering at the *Technische Universität Wien*, where he received the degree of *Diplomingenieur* in 2004. He is currently working on his doctoral degree. His research activities include circuit and device simulation, device modeling, and software development.

**SIEGFRIED SELBERHERR** was born in Austria in 1955. He received the degree of *Diplomingenieur* in electrical engineering and the doctoral degree in technical sciences from the *Technische Universität Wien* in 1978 and 1981, respectively. Dr. Selberherr has been holding the *venia docendi* on computer-aided design since 1984. Since 1988 he has been the chair professor of the Institute for Microelectronics. His current research interests are modeling and simulation of problems for microelectronics engineering.