International Journal of Parallel, Emergent and Distributed Systems
Vol. 24, No. 6, December 2009, 539–549

Taylor & Francis
Taylor & Francis Group

Synergies in scientific computing by combining multi-paradigmatic languages for high-performance applications

Philipp Schwaha*, René Heinzl, Franz Stimpfl and Siegfried Selberherr

Institute for Microelectronics, TU Wien, Gußhausstraße 27-29, Vienna, Austria

(*Received 8 January 2009; final version received 19 January 2009*)

The challenging art of multi-paradigmatic application development, which only few languages currently support, greatly aids the development of highly efficient and reusable software components. A link of two such languages, namely Python and C++, is presented. Thereby data structures and algorithms realised in C++ using features such as compile-time meta-programming are made available to the run-time environment of Python. Several generic components and modules for application design in the area of scientific computing are presented. Compile times and run-times are discussed to show the advantages of the proposed combination of both languages.

Keywords: high performance programming; scientific computing; multi-paradigmatic programming; Python; C++

1. Introduction

High-performance application design for scientific computing has always been a challenge regarding the utilised programming paradigms. An additional development in recent years regarding interpreted languages such as Python [30] raised the complexity in application design, but also introduced new possibilities, especially concerning rapid prototyping [31]. The Python programming language which has enjoyed a growing community of developers and users due to the easy availability of several different programming paradigms within a single language and its, by design, simple syntax and semantics, has many of the required features but lacks the essential feature of consistent and efficient traversal of simulation domains as well as quantity storage mechanisms. A key feature of Python is the ability to rapidly develop applications, efficiency however is limited by the interpreter, and the performance of the individual modules. The absence of multi-threading capabilities and inherent parallelisation is another severe issue currently not rigorously treated. Creating a high-performance link between the convenient Python control and parallel execution paths is therefore of utmost importance, as presented here [20]. While the capabilities of the interpreter continue to evolve, it is of high importance to provide efficient modules in order to insure acceptable run-times. This is especially true for the field of scientific computing, where processing of considerable amounts of data is commonplace.

Python modules are often based on highly optimised libraries written in a variety of compiled languages which are made available to Python by a wrapping layer. The multi-paradigmatic nature of both C++ and Python makes combinations of these two

* Corresponding author. Email: schwaha@iue.tuwien.ac.at

ISSN 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760902758552
http://www.informaworld.com

languages very appealing. Not only does C++ offer support for several paradigms concurrently, it also provides meta-programming mechanisms to orthogonally optimise and even parallelise developed code without modification. However, the great multitude of features and possibilities offered by C++ along with its strong typing mechanisms is often perceived as an obstacle, especially by beginners to programming, for rapid implementations of prototypes. On the other hand, Python has become an easy language to learn for rapid prototyping also in the field of scientific computing.

2. Combining multi-paradigmatic languages

It is a current trend to combine several programming languages, resulting in multi-language applications [25]. Different languages are utilised, each within the field where they perform best. Languages such as Python are then used to connect different modules. However, problems with interface specification and implementation arise with the combination of several programming languages, further complicating matters. In addition, the handling of different languages on different platforms becomes even more difficult.

In the field of scientific computing, the performance aspects should be handled orthogonally to the development of applications. Optimisations can thereby be treated separately. With the multi-language approach, performance aspects cannot be easily considered orthogonally because of the use of compiled modules which require an interface layer in order to build applications.

To enable the efficient transition between different languages, the paradigm and performance border across languages have to be resolved. An example is given by *Typhoon* which is presented here as a translation module from C++ to Python, as schematically shown in Figure 1.

Typhoon enables this transition by using the generic programming paradigm implementation of C++: static parametric polymorphism. The excellent run-time performance of compiled and highly-optimised code is thereby available in Python to allow rapid prototyping, with the multi-paradigmatic nature of Python making this transition especially simple and powerful by offering the utmost freedom for development.

These languages were selected by the availability of several mechanisms to use various programming paradigms, such as imperative, functional and generic programming. This is an essential feature in developing high-performance scientific applications. Another important fact is to develop reusable software components, where a software component is reusable if it can be used beyond its initial use within a single application or group of applications without modification.

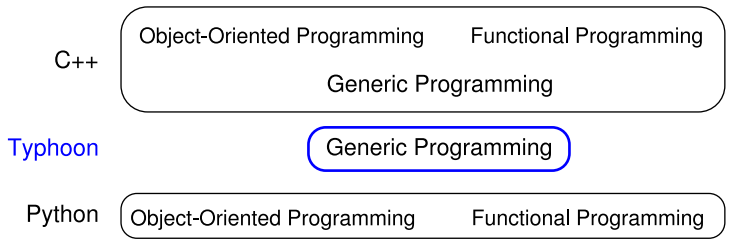

Figure 1. By enabling the transition from a statically typed language like C++ by *Typhoon*, the excellent run-time performance of compiled application parts can be adopted into an interpreted environment such as Python.

Classical languages, such as different types of assembler or C, use a straight-forward monomorphic programming style based on the imperative programming paradigm. In the field of scientific computing, only one- and two-dimensional data structures were initially used to develop applications, due to the limitations of computer resources. The imperative programming paradigm was sufficient for this type of task. Code that is developed this way supports only the initial data types and cannot be reused at all.

With the improvement of computer hardware and the rise of the object-oriented programming paradigm, the shift to more complex data models was possible. Modern high-level languages such as C++ or Java [15] implement means for polymorphic programming which make code reuse possible, e.g., by inheritance. Whereas several new programming languages such as Ruby [40] and other derivatives do not bother the user with the data types and therefore offer various automatically casted types.

Implications to application development can be observed clearly by studying the evolution of the object-oriented paradigm from imperative programming. The object-oriented programming paradigm has significantly eased the software development of complex tasks, due to the decomposition of problems into modular entities. It allows the specification of class hierarchies with its virtual class polymorphism (subtyping polymorphism), which has been a major enhancement for many different types of applications, but another important goal in the field of scientific computing, *orthogonal libraries*, cannot be achieved easily by this paradigm [9]. A simple example for an orthogonal library is a software component, which is completely exchangeable, e.g., a sorting algorithm for different data structures. An inherent property of this paradigm is the divergence of generality and specialisation [2,4,37].

Thus, the object-oriented programming paradigm is pushed to its limits by the various conceptual requirements in the field of scientific computing, due to problems with interface specifications, performance issues and lack of orthogonality. Even though the trend of combining algorithms and data structures is able to provide generalised access to the data structures through objects, it is observable that the interfaces of these objects become more complex as more functionality is added. Thus, the intended generality often results in inefficiency of the programs, due to virtual function calls, which have to be evaluated at run-time. Compiler optimisations such as inlining or loop-unrolling cannot be used efficiently, if at all. A lot of research has been carried out to circumvent these issues [6], but major problems arise in the details [13].

Functional programming is a radically different programming paradigm, where computation is treated as the evaluation of functions based on the following properties:

- Objects are provided only as constants or as expressions on objects.
- Functional programming avoids states of objects and mutable data.

The drawback of this paradigm is that it is already detached from the conventional programming style of C, Java, C++ or C# [5]. The functional modelling is fully reusable, because no assumptions about the used data types are made. Pure functional programming languages, such as Haskell, are completely polymorphic, concept-based programming languages. However, the issues with these languages are briefly explained by two drawbacks. First, the compiler has to do a lot of optimisation work to reach the excellent performance of other programming languages such as Fortran or C++ [12,29,36]. Second, some tasks, such as input/output operations are inherently not functional. These parts of an application are always given by different states due to the fact that data sources, e.g., hard-disk or memory, are state-based and can be modelled by the concept of a monad to introduce some kind of structure for functional programming [38,39].

Other developments regarding programming paradigms can be observed by the emerged generic programming paradigm [22,26] which has the same major goals as object-oriented programming, such as reusability and orthogonality. However, the problem is tackled from a different point of view [14]. Together with meta-programming [1], generic programming accomplishes both a general solution for most application scenarios and highly specialised code parts for minor scenarios without sacrificing performance [3,16,23] due to partial specialisation. The C++ language supports this paradigm with a type of polymorphism which is realised through template programming [33], static parametric polymorphism. Combining this type of polymorphism with meta-programming, the compiler can generate highly specialised code without adversely affecting orthogonality. This allows the programmer to focus on libraries which provide concise interfaces with an emphasis on orthogonality, as can already be found, e.g., in the BGL [34]. Although Java has gained more functionality with respect to a multi-paradigm approach [10,27], its performance still cannot be compared to the run-time performance of C++ [24]. Another comparison of high-level languages as well as scripting languages is also available [28].

3. Multi-paradigm development in C++

The developed generic scientific simulation environment (GSSE [18,19]) incorporates various functional as well as generic modules not only to support the close interaction with Python, but also the interoperability with the C++ STL and parallel STL [35], BGL [34], GrAL [7] and CGAL [11].

Basic data structures, such as the STL containers, already model a simple topological space and hence provide elementary topological traversal mechanisms. More complex data structures, e.g., a one-dimensional graph, can also be traversed by simple mechanisms, e.g., all vertices, all edges, vertex on edge and edge on vertex. Here, the distinction between so-called intrinsic traversal (vertices within a container) and deduced traversal (edge on vertex traversal) is important. Higher dimensional topological spaces, e.g., two- and three-dimensional meshes and grids, require a more complex combinatorial traversal hierarchy.

By providing a formal and common traversal interface for different types of libraries, interoperability is significantly enhanced. An example of using STL data structures with GSSE [17,18] concepts, e.g. an array, is presented next. Higher dimensional topological objects, such as edges, facets or cells, are not available directly in STL containers.

```
traverse<vertex>()  [  quan = quan_gen(1) ] (container);
```

STL type traversal of a container in C++

More complex traversal on a higher dimensional space can be accomplished in the following way for several libraries, in this case STL, CGAL, GrAL and GSSE, where all geometrical points with a special coordinate functor are marked:

```
traverse<segment>()
[
  traverse<vertex>()
  [
    if_(coord[x] > 5.3) [  quan = 1  ]
  ]
](domain);
```

Complex traversal of a container in C++

This example can be executed on an arbitrary number of cores due to the functional specification. Only a simple recompilation step with, e.g., the parallel STL is required.

4. The GSSE::Typhoon module

The advanced traversal and quantity storage mechanism of the GSSE is available to application developers directly in C++. For rapid prototyping or for data manipulation required during simulation set-up, a means of utilizing the same feature set flexibly without the need of recompilation may be desirable. To meet this demand, *Typhoon* links these mechanisms to the Python programming language, thereby enabling the highly efficient and parallel multi-dimensional and multi-topological traversal of the GSSE for the run-time scheme. The *Typhoon* Python module has been implemented using Boost Python [8], which simplifies the interfacing of C++ and Python. Care has to be taken to correctly transfer the high flexibility awarded to the GSSE by employing several programming paradigms in concert with the ones available to Python. A particular difficulty is the fact that the static polymorphism used in C++ for performance and consistency reasons must be transferred to the dynamically typed world of Python. While generic programming techniques are used to minimise the implementation effort, the resulting compile times cannot be neglected, as all desired facilities for all required dimensions must be instantiated at compile time in order to be available at run-time. In a C++ application, the correct dimension is automatically selected at compile time, while in the case of Python applications of dimensions one to three, the single code base results in three separate Python modules, with their proper selection automatically performed by Python's dynamic type system and by function overloading.

The following short code snippet demonstrates the application of the traversal mechanisms, where the same traversal mechanism is used as in the C++ example. First all segments in a domain are traversed, followed by the traversal of the cells of the traversed segment. A quantity is stored on all of the traversed cells using the identifier 'quan_1'. Then a sample code is given to present, how the traversal mechanisms can be combined with Python's lambda function facilities to obtain a powerful selection mechanism. The result of such a selection is again compatible with *Typhoon*'s facilities, as is shown in the last two lines of code.

```
for segment in segments(domain):
    for element in cells(segment):
        store_cell_quan(domain,segment,element,"quan_1", 1.0)

    selection = filter(lambda x:
                       filter (lambda y: coordinates(d,y)[0] > 5.3,
                               vertices(x)),
                       cells(segment))

    for selected in selection:
        sum = 0.0
        for v in vertices(selected)
              sum += retrieve_vertex_quan(domain, v,"quan_3")

        store_cell_quan(domain,segment,selected,"quan_2",
                        sum / len(vertices(selected)))
```

Using *Typhoon* for traversal and quantity access.

Here, the actual traversal is executed by the GSSE traversal library, where the control of run-time selections is handled by *Typhoon.*

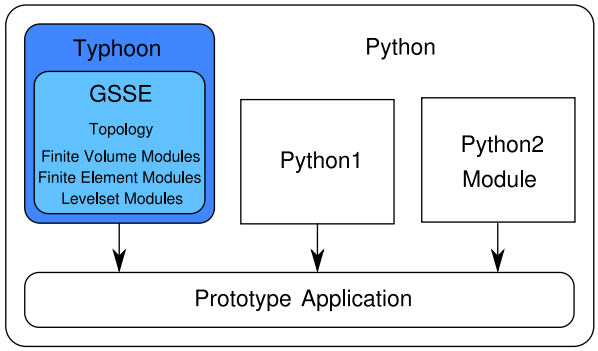

Figure 2. The *Typhoon* module brings the topology and traversal mechanisms of the GSSE to Python. The combination with additional Python modules enables a rapid implementation of prototype applications.

Using the *Typhoon* Python module it is possible to also rapidly develop application prototypes by combining it with one of the numerous scientific packages already available for Python. Figure 2 schematically shows the interaction of the GSSE, *Typhoon*, and Python.

The availability of a wide variety of traversal mechanisms in conjunction with high performance solver packages such as Trilinos [21], which is made available in Python by PyTrilinos [32], is a particularly interesting combination, as it enables even complete implementations of simulations directly in Python. Typically, the input is used to assemble an equation system which is subsequently solved to compute an approximate solution of the problem under investigation. By using the *Typhoon* module, the Python code remains unchanged even for different topologies and dimensions, as the underlying GSSE takes care of equalizing the interfaces.

By traversing the vertices of the input structure the system matrix is assembled using *Typhoon*. The high level of abstraction provided by the GSSE is retained without restrictions on the topology or dimension of the specified problem. In contrast to a C++ implementation, where the, often time consuming, recompilation of the program is required to obtain an executable from the source code, the Python implementation is available immediately. A caveat, however, is that the *Typhoon* module had to be compiled including the appropriate dimension and topology, which is automatically taken care of by C++ during compilation.

The following source code demonstrates the combination of *Typhoon*'s traversal mechanisms, which are used to assemble the matrix, and the PyTrilinos solver interface, which is employed to calculate the solution:

```
for segment in segments(domain):
    for element in vertices(segment):
        assign_matrix_value(matrix_A, element) = calc(element)
        assign_matrix_value(matrix_B, element) = calc_rhs(element)
Solver.SetMatrix(matrix_A)
Solver.SetVectors(X, matrix_B)
Solver.Solve()
```

Using *Typhoon* and PyTrilinos to assemble a matrix.

4.1 Technical details

Making the template heavy traversal mechanisms offered by GSSE compatible with the requirements of the dynamic run-time requires the instantiation of all the required code parts. A helper construct, which encapsulates the necessary steps has been developed to prohibit the writing of excessive amounts of code and to ease use. A visualisation of the general architecture is given in Figure 3. The interplay of both, the topological structure, which describes the data structures, as well as the algorithmic aspects of the GSSE are mapped to Python using a single entity, thereby ensuring consistency. As has been shown in the previous examples, the exposure is also compatible with the specification of algorithms in Python using the functional programming paradigm.

The main entity containing all subsequent type definitions and helper functions is given in the following short piece of code along with a few basic type definitions. The template parameter `CCT` is the GSSE domain which is to be exported to Python. The following type definitions make various basic GSSE entities available internally.

```
template <typename CCT>
struct python_domain_glue
{   typedef typename gsse::CC_traits<CCT>::segment_type segment_type;
    typedef typename gsse::CC_traits<CCT>::vertex_type  vertex_type;
    typedef typename gsse::CC_traits<CCT>::edge_type    edge_type;
```

Generic programming in C++

The next code sample demonstrates, how simple helper functions are employed to extract instances of the required meta-programmed GSSE constructs, with this particular case dealing with traversal.

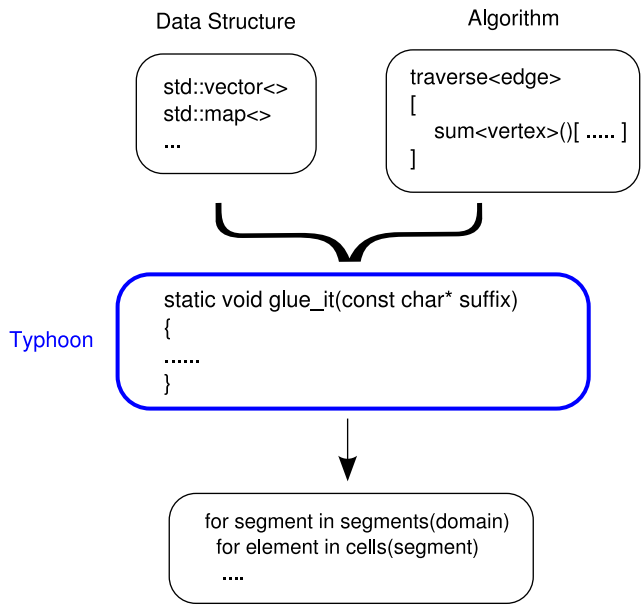

Figure 3. Using *Typhoon* the interplay of data structures and the algorithms working on them is translated using the datatype system of the C++ compiler to automatically create high-performance object code. The resulting module is then available in Python and can be used with the programming paradigms available in Python.

```
static typename intrinsic_traversal<ElementT, IterationT>::type
                fetch_elements(ElementT& element)
{ return typename
  intrinsic_traversal<ElementT, IterationT>::type(element);
}
```

Generic programming in C++

The pieces of code shown are used to facilitate the export using Boost Python. The export definitions themselves are contained in a single wrapping function for ease of use.

```
static void glue_it(const char* char_suffix)
{  const std::string suffix(char_suffix);
   bp::class_<CC_type> (("CC_" + suffix).c_str())
      .def_readonly("vertex", vertex_type())
      .def_readonly("edge",   edge_type())
      .def("__getitem__", get_element_quan<vertex_type>)
      .def("__getitem__", get_element_quan<edge_type>)
      .def("__setitem__", assign_data);
   ...
   boost::python::class_<segment_type> (("segment_" + suffix).c_str());
```

Wrapped export statements using Booost Python.

The `suffix` used in the function has to be unique for each GSSE domain exported, as it is used to remove ambiguity for the dynamic type system. The next piece of code demonstrates the final binding of the traversal helper function to the internal types and its export to Python.

```
   bp::def("vertices", fetch_elements<segment_type, vertex_type>);
   bp::def("edges",    fetch_elements<segment_type, edge_type>);
   ...
}
```

Binding of traversal the function to internal types.

By employing the same technique for all types derived from a GSSE domain, every invocation of an export directive is completely self-contained. The construct not only defines types derived from the specified domain, but also ensures that the required functionality is made available for these types. This is accomplished by assigning internal names to the required types and functions. At the same time, a consistent interface to Python is created, which depends on the internal types, thereby being resolvable by the dynamic type system at run-time. It is thereby possible to make the encapsulated functionality available to the user as easily as possible.

It should also be noted, that since GSSE provides a super-set of facilities to the STL standard containers and algorithms, the introduced technique can also be applied to STL containers and algorithms.

5. Benchmarks

The compile times of associated with the structures required for rigorous traversal operations cannot be neglected, especially in higher dimensions. Table 1 gives an overview of the time consumed for compilation for several dimensions. While *Typhoon* cannot alter the compile times themselves, it serves to save a lot of time as compilation

Table 1. Comparison of compile times for various dimensions.

Dimension	1	2	3	4	5
Compile time	1.07 s	1.47 s	11.99 s	971.26 s	>20 h

Table 2. Comparisons of the traversal times of data structures from Python and *Typhoon* (times obtained on an AMD Phenom 9600).

No. of elements	10^8	$10{,}000 \times 10{,}000$	$100 \times 1000 \times 1000$
Python	9 m 26 s	3 m 35 s	3 m 57 s
Typhoon	2 m 44 s	1 m 18 s	2 m 29 s

needs to be performed but once for the facilities to be available in Python without further compile time delays, whereas the time has to be expended repeatedly in the case of C++.

To further compare the two different approaches controlled by *Typhoon*, we give benchmark results for topological traversal. The benchmarks are obtained by simple traversal of an array as equivalent data structures are available in Python and *Typhoon*. No quantities were stored on the traversed objects. It should be noted that memory consumption was much higher in native Python than when using *Typhoon* and even prohibited the traversal of more than the given 10^8 elements.

The second and third columns are for multi-dimensional arrays and show that *Typhoon* is always faster when traversing multi-dimensional structures than Python. Again Python's native memory requirements surpassed those of *Typhoon*.

The memory issue is expected to become even more pronounced, when quantities are to be stored on the traversed structures as *Typhoon* inherently makes use of the GSSE's quantity handling capabilities (Table 2).

6. Conclusion

A module linking the two multi-paradigmatic languages C++ and Python named *GSSE::Typhoon* has been presented. The multi-dimensional and multi-topological capabilities of the C++ framework GSSE are automatically translated to Python for a desired dimension and topology, thus making the highly optimised and parallelisable traversal and quantity storage facilities available to Python. It thereby provides a feature set important to scientific computing which native Python lacks. The compatibility of the traversal mechanisms with Python's multi-paradigmatic nature as well as other Python software packages results in a powerful tool for scientific computing, useful for both rapid prototyping and simulation setup. Furthermore, the employed concepts, are not limited to the GSSE, but are also easily applicable to the wide range of STL standard containers and algorithms.

Acknowledgement

This work has been supported by the Austrian Science Fund FWF, project P19532-N13.

References

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, Addison-Wesley, Boston, MA, 2004.

[2] R. Affeldt, H. Masuhara, E. Sumii, and A. Yonezawa, Supporting objects in run-time bytecode specialization, in *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ACM Press, New York, NY, 2002, pp. 50–60.
[3] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, Boston, MA, 2001.
[4] H.M. Andersen and U.P. Schultz, *Declarative specialization for object-oriented-program specialization*, in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ACM Press, New York, NY, 2004, pp. 27–38.
[5] T. Archer and A. Whitechapel, *Inside C#*, 2nd ed., Microsoft Press, Redmond, WA, 2002.
[6] D.F. Bacon, S.L. Graham, and O.J. Sharp, *Compiler transformations for high-performance computing*, in *Proceedings of the Object-oriented Programming Systems, Languages, and Applications Conference*, Vol. 26, 1996, pp. 345–420.
[7] G. Berti, *GrAL – The grid algorithms library*, in *Proceedings of the Computational Science ICCS*, Vol. 2331, Springer, London, 2002, pp. 745–754.
[8] Boost: Boost Python (2006), http://www.boost.org/
[9] K. Bruce, L. Cardelli, G. Castagna, G.T. Leavens, and B. Pierce, *On binary methods*, Theor. Pract. Object Syst. 1(3) (1995), pp. 221–242.
[10] A. Dekker, *Lazy functional programming in Java*, ACM Sigplan Not. 41(3) (2006), pp. 30–39.
[11] A. Fabri, *CGAL – The computational geometry algorithm library*, in *Proceedings of the 10th International Meshing Roundtable*, CA, USA, 2001, pp. 137–142.
[12] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock, *An extended comparative study of language support for generic programming*, J. Funct. Program. 17(2) (2007), pp. 145–205.
[13] D. Gay and B. Steensgaard, *Fast escape analysis and stack allocation for object-based programs*, in *CC '00: Proceedings of the 9th Conference on Compiler Construction*, Springer, London, 2000, pp. 82–93.
[14] T. Geraud and A. Duret-Lutz, *Generic programming redesign pattern*, in *Proceedings of the 5th Conference on Pattern Languages of Programs (EuroPLoP 2000)*, Irsee, Germany, 2000.
[15] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed., Addison Wesley, Boston, MA, 2005.
[16] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp, *Generic programming and high-performance libraries*, Intl J. Parallel Prog. 33(2) (2005), pp. 145–164.
[17] R. Heinzl and P. Schwaha, *GSSE* (2007), http://www.gsse.at/.
[18] R. Heinzl, P. Schwaha, and S. Selberherr, *A high performance generic scientific simulation environment*, in B. Kaagström, E. Elmroth, J. Dongarra and J. Wasniewski, eds., Lecture Notes in Computer Science, Vol. 4699, Springer, Berlin, 2007, pp. 781–790.
[19] R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr, *A parallel generic scientific simulation environment*, in *Proceedings of the PARA Conference*, Trondheim, Norway, May, 2008.
[20] R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr, *Parallel library-centric application design by a generic scientific simulation environment*, in *Proceedings of the POOSC Conference*, Paphos, Cyprus, July, 2008.
[21] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, et al., *An overview of the Trilinos project*, ACM Trans. Math. Software 31(3) (2005), pp. 397–423.
[22] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J.G. Siek, *Algorithm specialization in generic programming – Challenges of constrained generics in C++*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, New York, NY, 2006, pp. 272–282.
[23] J. Järvi, J. Willcock, and A. Lumsdaine, *Concept-controlled polymorphism*, in *GPCE '03: Proceedings of the 2nd Conference on Generative Programming and Component Engineering*, Springer, New York, NY, 2003, pp. 228–244.
[24] I. Kazi, H. Chen, B. Stanley, and D. Lilja, *Techniques for obtaining high performance in Java programs*, ACM Comput. Surveys 32(3) (2000), pp. 213–240.
[25] H.P. Langtangen and X. Cai, *Mixed language programming for HPC applications*, in *Proceedings of the PARA Conference*, Umea, Sweden, June, 2006, p. 154.
[26] D.R. Musser and A.A. Stepanov, *Generic programming*, in *Proceedings of the ISSAC'88 on Symbolic and Algebraic Computation*, Springer, London, 1988, pp. 13–25.
[27] M. Naftalin and P. Wadler, *Java Generics and Collections*, O'Reilly & Associates, Sebastopol, CA, 2006.

[28] L. Prechelt, *An empirical comparison of seven programming languages*, Computer 33(10) (2000), pp. 23–29.
[29] A. Priesnitz, *Multistage algorithms in C++*, Dissertation, Universität Göttingen, 2005.
[30] Python Software Foundation, *Python programming language*. Available at http://www.python.org/.
[31] C. Rickett, S. Choi, C. Rasmussen, and M. Sottile, *Rapid prototyping frameworks for developing scientific applications: A case study*, J. Supercomput. 36(2) (2006), pp. 123–134.
[32] M. Sala, W. Spotz, and M. Heroux, *PyTrilinos: High-performance distributed-memory solvers for Python*, ACM Trans. Math. Software 34(2) (2008), pp. 1–33.
[33] J.G. Siek and A. Lumsdaine, *Concept checking: Binding parametric polymorphism in C++*, in *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.
[34] J. Siek, L.Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley, Boston, MA, 2002.
[35] J. Singler, P. Sanders, and F. Putze, *The multi-core standard template library*, in Lecture Notes in Computer Science, Vol. 4641, Springer, Berlin, 2007, pp. 682–694.
[36] T.L. Veldhuizen, *Expression templates. C++ Report*, 7(5) (1995), pp. 26–31, reprinted in C++ Gems, ed. Stanley Lippman.
[37] E.N. Volanschi, C. Counsel, G. Muller, and C. Cowan, *Declarative specialization of object-oriented programs*, in *Proceedings of the Object-oriented Programming Systems, Languages, and Applications Conference*, ACM Press, New York, NY, 1997, pp. 286–300.
[38] P. Wadler, *Monads for functional programming*, LNCS 925 (1995), pp. 24–52.
[39] P. Wadler, *How to declare an imperative*, ACM Comput. Surveys (CSUR) 29(3) (1997), pp. 240–263.
[40] R. Wirdemann and T. Baustert, *Rapid Web Development mit Ruby on Rails*, Hanser, München, 2006.