The Forced Evolution of Implementations

Using a Monte Carlo Algorithm as Example

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

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

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

ABSTRACT

Scientific computing has seen very rapid growth during its relatively young existence. The continued growth, however, is often impeded by neglecting the advancements of programming paradigms and compiler technologies. We present an approach which not only capitalizes on new developments, but also reuses already existing code bases using a Monte Carlo algorithm as an example.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; I.6 [**Computing Methodologies**]: SIMULATION AND MODELING

Keywords

Scientific computing, high performance computing, Monte Carlo, Boltzmann equation, generic programming

1. INTRODUCTION

The use of simulations in the field of sciences is increasing in frequency and importance, as it provides a means of evaluating the feasibility of evolving scientific models. The increasing complexity of new theories and models makes this an invaluable tool, which complements simpler, purely theoretical considerations, for the pre-selection of setups of often very cost intensive experiments. However, the rising complexity of problems to be addressed by simulations constantly requires the use of efficient numeric algorithms and implementations. At the same time scientific computing continues, further motivated by the relentless hunger for computational power, to develop and improve implementation techniques, programming paradigms, hardware platforms as well as compiler technologies. This, however, has resulted in the growth of a gap between the domain experts deploying simulations and scientific computing. As a result many simulation codes available as well as newly developed only use very basic programming techniques and paradigms and do not take advantage of their continued evolution. This does not come as a surprise considering that domain experts view implementations to be of secondary importance and a necessary evil, and newly developed programming paradigms as exotic at best. The resulting prevailing procedural programming style, which provides very little room for higher abstractions, also results in a high level of conservative attitude when choosing the language for implementation for simulation software, still making Fortran and C the prevailing languages in, e.g., the area of physics or engineering.

However, the low level of abstraction used in these codes increases the amount of code which has to be written, as reusability, which is not commonly a goal in this setting, is usually low. At this time, the domain expert, software designer, programmer, tester, and end-user is one person, a situation which complicates and slows down the development process as a whole, as the source code is often only poorly commented or documented in other ways. Therefore once the initial developers have moved on to different tasks, reconstruction of the intended semantics of the low level implementation becomes extremely costly. This more often than not results in complete abandonment of already tried, tested and often tediously debugged implementations for further developments, as the resource requirements for such endeavors become prohibitive in favor of completely new redevelopments. A step such as this necessarily requires the allocation of considerable resources and not only abandons the development of the old code base, but also much of the experience associated with it.

It is for this reason, that a multitude of software applications and tools, which provide methods and libraries for the solution of very specific problem classes, has been developed. However, they are mostly specialized for a certain type of underlying mathematical model. Only in the recent past have environments for various highly differing problems been developed and published, all with their advantages and disadvantages. However, it is important to note that applications not developed with interoperability in mind impose restrictions on possible solution methods which can not be foreseen at the beginning of program development.

It is therefore highly desirable to rejuvenate the implementation which is already available so that it utilizes advanced technologies and techniques while at the same time keeping as much of the already obtained experience and trust related to the original code base. It is therefore proposed to approach this task in an evolutionary fashion initially including as much of the old implementation as possible and gradually replacing it to bring it up to date to what may appear as revolutionary when compared to the initial techniques. Such a procedure is only made feasible by using an environment based on concepts.

*Contact information: schwaha@iue.tuwien.ac.at
†Contact information: heinzl@iue.tuwien.ac.at
‡Contact information: nedialkov@iue.tuwien.ac.at

POOSC '09, July 7 2009, Genova, Italy

2. GENERIC ENVIRONMENT

The combination of different programming paradigms fits the scenario of scientific computing exceptionally well. The generic programming paradigm establishes homogeneous interfaces between algorithms and data structures without sub-typing polymorphism. Functional programming eases the specification of equations and offers extendable expressions while retaining the functional dependence of formulae by higher order functions. Also, this type of specification of access, algebraic manipulation, and traversal circumvent the problems of an imperative implementation. The features of meta-programming offer the embedding of domain-specific terms and mechanisms directly into the host language as well as compile time algorithms to obtain optimal run time. Developments toward an alternative compilation model and active library design are also an important step [1, 2]. However, reusability of traditional application parts and even libraries is often extremely limited due to the following issues:

- *Numerical data types.* There are numerous well-known numerical data types which also are often optimized for special applications in order to yield high performance. Only with generic interfaces can these performance-enhancing measures be used in different kinds of applications.
- *Topologies.* Numerical schemes often require different underlying topological data structures. While some applications perform well using structured grids, other applications require unstructured meshes with varying local feature sizes. Although the nature of these topologies is totally different, standardized interfaces for all topological data types have to be provided.
- *Different dimensions.* Special symmetries that are encountered in many problems of scientific computing can be used to reduce the effective dimension of a calculation. Even though all problems can be treated in their full dimension, an enormous gain in performance by using lower dimensional data structures can not be neglected.
- *Equation system assembly.* Most of the solver mechanisms require an initialization of the values of their own interfaces. Therefore, an interface which abstracts these specialties and makes the solvers accessible in a general manner is required. With such an interface the governing equations can be formulated independently of the actual data structures of the solver.
- *Solution of large equation systems.* A lot of problems result in large equation systems which have to be solved. There are solvers available for various special cases, which perform well under certain circumstances but fail to converge sometimes. Therefore, interface design has to guarantee that different solvers can be used.

To circumvent the stated issues, a set of requirements for library-centric application design in the field of scientific computing is given in the following to allow the transformation of the concepts for scientific computing into generically applicable and efficient software components. The achieved library-centric design is facilitated, as the following criteria are met:

- The environment is complete, so all applications can be written exclusively using its libraries (as well as standard libraries). Indeed, completeness increases usability enormously, because no components have to be added while existing components can be adapted.
- The components of the environment are usable for a broad range of different applications.
- The interoperability of the environment is not affected by its completeness. Even though all the libraries can be used by themselves, they provide standardized interfaces, which guarantees compatibility for data structures which have not been foreseen in the initial design.

An additional requirement for application design is related to an efficient use of programming paradigms. A basic layer can be identified, which has to implement a formal topological interface for container properties as well as for traversal, thereby establishing a consistent interface for data structures and quantity storage. The object-oriented and generic programming paradigms are best suited to accomplish this. On top of this layer, functional expression specification facilities are required, which can be modeled best by the functional programming paradigm. The concept of a domain-specific embedded language (DSEL) in C++ requires the additional concept of meta-programming resulting in an active library concept to implement and guarantee an overall high performance [3].

In this particular case the generic scientific simulation environment (GSSE) [4–6] is used which was specifically designed to address the described issues and meat the presented requirements for scientific computing. Therefore it is easy to build virtually any kind of simulation or scientific software using the components of the GSSE.

3. MONTE CARLO APPLICATION

Monte Carlo simulations are a power-full, yet computationally expensive numerical tool with seemingly unlimited areas of application [7]. The high numerical cost have always made researchers very keen on utilizing every optimization available. For low level codes this often results in portability issues as every different platform may require drastically different implementations to perform well, as the platform specifics, which should be accounted for by the compiler automatically, are done manually. The use of the generic environment GSSE and the possibilities it provides allows to optimize the implementation even when the platform on which it is to be run changes while ensuring a high level of abstraction.

The following will use the Monte Carlo algorithms from the field of semiconductor simulations used to calculate solutions to Boltzmann's equation for electrons, which is popularly given in the form:

$$\left[\frac{\partial}{\partial t} + \mathbf{v}\,\mathrm{grad}_{\mathbf{q}} + \mathbf{F}\,\mathrm{grad}_{\mathbf{p}}\right] f = Q\,(f) \tag{1}$$

Where the left hand side describes classical Newton trajectories including the electric field and the right hand side models interactions of an electron with the semiconductor crystal.

Only the trajectories traversing the simulation domain actually depend on geometry and the associated quantity complex. Therefore the implementation of this part makes use of the GSSE, which provides high performance operations for virtually arbitrary dimensions and topologies at high run time performance.

In contrast to this, the collision term on the right hand side does not depend on the geometric properties, but only makes use of the current state of the electron (its position and its momentum) along with a quantity complex, which is also provided by the GSSE, and a collection of physical collision models.

To facilitate treatment using the Monte Carlo method Equation (1) is usually reformulated, as is also done in this particular example. An algorithm has been developed to calculate the correction of quantum mechanical results due

to classical scattering interactions [8, 9]. The problem takes the form of a Fredholm integral equation of the second kind and is then further broken down to integrals such as:

$$\int \left\{ \int f(x')S(x', x)dx' - f(x)\lambda(x) \right\} e^{-\int_t^0 \lambda(x(\tau))d\tau} dt$$

where x is a point in the phase space and S and λ describe the scattering and which can be evaluated using the Monte Carlo approach using a rejection technique.

The rejection technique is shortly sketched in Figure 1. At a given point x a *current rate* is chosen randomly using a distribution from the interval $[0; 1[$ multiplied by an upper bound for all the scattering. Then the scattering models are evaluated at this point x and the individual rates, thusly calculated are summed in order to determine the interval λ_i which contains the *current rate*. The scattering mechanism associated with this interval then chooses the new state of the electron.

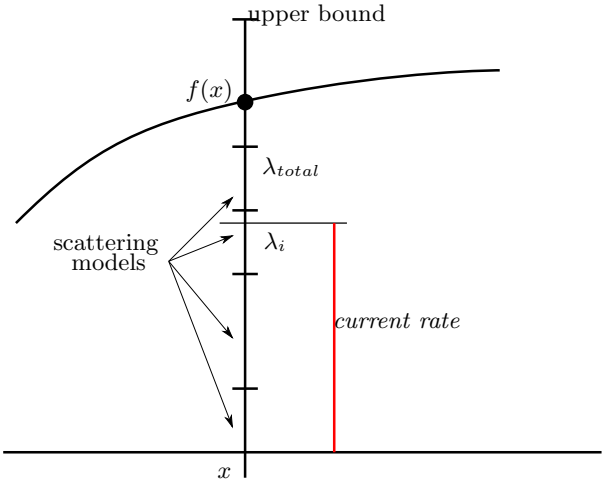

Figure 1: Schematic illustration of the used Monte Carlo aglorithm.

This shows the following requirements to implement this algorithm:

- A reliable random number generator.
- A rate has to be determinable for every scattering mechanism.
- A transition from an initial state to a final state has to be available for every scattering mechanism.
- An upper bound needs to be available for the function $f(x)$, where it is possible to choose a constant, but more efficient, when employing polynomial functions.

During the simulation the scattering models are evaluated repeatedly in order to choose the next state of the electron under consideration. Therefore high performance is among the requirements of the collision model.

3.1 Legacy Application

The legacy application has been write in plain ANSI C. All available scattering mechanisms are implemented as individual functions, which are called subsequently. The scattering models require a variable set of parameters, which leads to non-homogeneous interfaces in the functions representing them. To alleviate this to some extent global variables have been employed completely eliminating any aspirations of data encapsulation and posing a serious problem for attempts for parallelization to take advantage of multi-core CPUs. The code has the very simple and repetitive structure:

```
double sum = 0;
double current_rate = generate_random_number();

if (A_key == on)
{
  sum = A_rate(state, parameters);

  if (current_rate < sum)
  {
   counter->A[state->valley]++;
   state_after_A (st, p);
   return;
  }
}

if (B_key == on)
{
 sum += B_rate (state, state_2, parameters);

 if (current_rate < sum)
 {
  counter->B[state->valley]++;
  state_after_B (state, state_2);
  return;
 }
}
...
```

Extensions to this code are usually accomplished by copy and paste, which is prone to simple mistakes by oversight, such as failing to change the counter which has to be incremented or calling the incorrect function to update the electron's state.

Furthermore, at times the need arises to calculate the sum of all the scattering models (λ_{total} in Figure 1), which is accomplished in a different part of the implementation, thus further opening the possibility for inconsistencies between the two code paths.

The decision which models to evaluate is done strictly at run time and it would require significant, if simple, modification of the code to change this at compile time, thus making highly optimized specializations very cumbersome.

The functions calculating the rates and state transitions, however, have been well tested and verified, so that abandoning them would be wasteful.

3.2 Evolved Application

Scientific computing requires not only high performance components evaluated and optimized at compile time, but also run time exchangeable physical models and the ability to cope with various boundary conditions. The two most commonly used programming paradigms, object oriented and generic programming, differ in how the required functionality is implemented.

Object oriented programming directly offers run time polymorphism by means of virtual inheritance. Unfortunately current implementations of inheritance use an intrusive approach for new software components and tightly couples a type and the corresponding operations to the super type. In contrast to object-oriented programming, current applications of generic programming are limited to algorithms using statically and homogeneously typed containers but offers highly flexible, reusable, and optimizeable software components.

As can be seen, both programming types offer different points of evaluation. Run time-polymorphism based on concepts [10] (run time concepts) tries to combine the virtual inheritance run time modification mechanism and the compile time flexibility and optimization.

Inheritance in the context of run time polymorphism is used to provide an interface template to model the required concept where the derived class must provide the implementation of the given interface. The following code snippet

```
template<typename StateT> struct scatter_facade
{
 typedef StateT state_type;

 boost::shared_ptr<scattering_concept>
  scattering_object;

 struct scattering_concept
 {
  virtual ~scattering_concept() {} ;
  virtual numeric_type
   rate(const state_type& input) const = 0;
  virtual void
   transition(state_type& input) = 0;
 };

 template<typename T> struct scattering_model :
  scattering_concept
 {
  T scattering_instance;
  scattering_model(const T& x) :
   scattering_instance(x) {}
  numeric_type
   rate(const state_type& input) const ;
  void transition(state_type& input) ;
 };

 numeric_type
  rate(const state_type& input) const;

 void transition(state_type& input) ;

 template<typename T>
  scatter_facade(const T& x) :
  scattering_object(new
    scattering_model<T>(x)) {}

 ~scatter_facade() {}
};
```

therefore introduces a `scattering_facade` which wraps a `scattering_concept` part. The virtual inheritance is used to configure the necessary interface parts, in this case `rate()` and `transition()`, which have to be implemented by any scattering model. In the given example the `state_type` is still available for explicit parametrization.

To interface this novel approach a core structure is implemented which wraps the implementations of the scattering models:

```
template<typename ParameterType>
struct scattering_rate_A
{
 ...
 const ParameterType& parameters;

 scattering_rate_A
 (const ParameterType& parameters):
  parameters(parameters) {}

 template <typename StateType> numeric_type
 operator() (const StateType& state) const
 {
  return A_rate(state, parameters);
 }
};
```

By supplying the required parameters at construction time it is possible to homogenize the interface of the `operator()`. This methodology also allows the continued use of the old data structures in the initial phases of transition, while not being so constrictive as to hamper future developments.

The functions for the state transitions are treated similarly to those for the rate calculation. Both are then fused in a `scattering_pack` to form the complete scattering model and to ensure consistency of the rate and state transition calculations and which also models the run time concept as can be seen in the following part of code:

```
template<scattering_rate_type,
         transition_type,
         parameter_type>
struct scattering_pack
{
 ...
 scattering_rate_type<parameter_type>
  rate_calculation;
 transition_type<parameter_type>
  state_transition;

 scattering_pack
  (const parameter_type& parameters) :
  rate_calculation(parameters),
  state_transition(parameters)
 {}

 template<typename StateType>
 numeric_type rate(const StateType& state) const
 {
  return rate_calculation(state);
 }

 template<typename StateType>
 void transition(StateType& state)
 {
  state_transition(state);
 }
}
```

The blend of run time and compile time mechanisms allows the storage of all scattering models within a single container, e.g. `std::vector`, which can be iterated over in order to evaluate them.

```
typedef std::vector<scatter_facade_t
 <state_type> >
  scatter_container_type ;
scatter_container_type scatter_container ;
scatter_container.push_back(scattering_model) ;
```

For the development of new collision models easy extendability, even without recompilations, is also a highly important issue. This approach allows the addition of scattering models at run time and to expose an interface to an interpreted language such as, e.g., Python [11].

In case a highly optimized version is desired, the run time container (here the `std::vector`) may be exchanged by a compile time container, which is also readily available from the GSSE and provides the compiler with further opportunities for optimizations at the expense of run time adaptability.

3.3 Incurred Benefits

While the described approach initially slightly increases the burden of implementation, due to the fact that wrappers need to be provided, it gives a transition path to integrate legacy codes into an up to date frame while at the same time not to abandoning the experience associated with it. The invested effort allows to raise the level of abstraction, which in turn allows to increase the benefits obtained from the advances in compiler technologies. This in turn inherently allows an optimization for several platforms without the need for massive human effort, which was needed in previous approaches.

In this particular case, encapsulating the reliance on global variables of the functions implementing the scattering models to the wrapping structures, parallelization efforts are

greatly facilitated, which are increasingly important with the continued increase of computing cores per CPU.

Furthermore the results can easily be verified as code parts are gradually moved to newer implementations, the only stringent requirement being link compatibility with C++. This test and verification can be taken a step further in case the original implementation is written in ANSI C, due to the high compatibility of it to C++. It is possible to weave parts of the new implementation into the older code. Providing the opportunity to get very a fine grained comparison not only of final results, but of all the intermediates as well.

Such swift verification of implementations allows to also speed up the steps necessary to verify calculated results with subsequent or contemporary experiments, which should not be neglected, in order to keep physical models and their numerical representations strongly rooted in reality.

4. CONCLUSION

An approach to reuse and port legacy implementations in conjunction with up to date programming techniques has been presented. It uses the already existing generic high performance framework GSSE as a foundation. Additionally run time concepts are employed to provide high flexibility at run time while also enforcing the adherence to the specified interfaces to satisfy requirements. By combining the features of run time and compile time methodologies the migration path of implementations to higher abstraction levels is able to exploit the advances in language tools such as compilers, which provide automated adaptations to various target platforms.

As it is not feasible to provide a monolithic software library suitable for each and every case in general, it is the goal to extract and refine generally useful components, which, due to the fact of keeping interoperability in mind, can be used to quickly and adaptively construct high performance solutions for various problems encountered in scientific computing.

5. ACKNOWLEDGMENTS

The authors wish to acknowledge Franz Stimpfl and Prof. Siegfried Selberherr for their support. This work has been supported by the Austrian Science Fund FWF, project 19532-N13.

6. REFERENCES

[1] T. L. Veldhuizen. Five Compilation Models for C++ Templates. In *1st Workshop on C++ Template Programming*, October 2000.

[2] T. L. Veldhuizen and D. Gannon. Active Libraries: Rethinking the Roles of Compilers and Libraries. In *Proc. of the SIAM Workshop on Obj.-Oriented Methods for Inter-Operable Sci. and Eng. Comp. (OO'98)*. SIAM, 1998.

[3] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser. Performance Aspects of a DSEL for Scientific Computing with C++. In *Proc. of the POOSC Conf.*, pages 37–41, Nantes, France, July 2006.

[4] R. Heinzl. *Concepts for Scientific Computing*. Dissertation, Technische Universität Wien, Austria, 2007.

[5] R. Heinzl, P. Schwaha, and S. Selberherr. A High Performance Generic Scientific Simulation Environment. In B. Kaagström et al., editor, *Lecture Notes in Computer Science*, volume 4699/2007, pages 996–1005. Springer, Berlin, June 2007.

[6] R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr. Parallel Library-Centric Application Design by a Generic Scientific Simulation Environment. In *Proc. of the POOSC*, Paphos, Cyprus, July 2008.

[7] I. Dimov. *Monte Carlo Methods For Applied Scientists*. World Scientific Publishing Company, 2005.

[8] M. Nedjalkov, P. Schwaha, O. Baumgartner, and S. Selberherr. Particle Model of the Scattering-Induced Wigner Function Correction. In *Proc. of the 7th LSSC*, Sozopol, Bulgaria, June 2009.

[9] P. Schwaha, O. Baumgartner, R. Heinzl, M. Nedjalkov, S. Selberherr, and I. Dimov. Classical Approximation of the Scattering Induced Wigner Correction Equatio. In *Proc. of the 13th IWCE*, pages 177 – 180, Beijing, China, May 2009.

[10] P. Pirkelbauer, S. Parent, M. Marcus, and B. Stroustrup. Runtime Concepts for the C++ Standard Template Library. In *Proc. of the ACM SAC'08*, pages 171–177, Fortaleza, Ceara, Brazil, March 2008.

[11] P. Schwaha, R. Heinzl, F. Stimpfl, and S. Selberherr. Synergies in Scientific Computing by Combining Multi-Paradigmatic Languages for High-Performance Applications. In *Proc. of the POOSC*, Paphos, Cyprus, July 2008.