Utilizing Modern Programming Techniques and the Boost Libraries for Scientific Software Development

Josef Weinbub
Institute for Microelectronics
Technische Universität Wien
Vienna, Austria
weinbub@iue.tuwien.ac.at

Karl Rupp
Institute for Microelectronics
and Institute for Analysis and
Scientific Computing
Technische Universität Wien
Vienna, Austria
rupp@iue.tuwien.ac.at

Siegfried Selberherr
Institute for Microelectronics
Technische Universität Wien
Vienna, Austria
selberherr@iue.tuwien.ac.at

ABSTRACT

Modern programming techniques and libraries provide software developers with a vast set of functionality and flexibility. However, applying the associated techniques like generic, functional, or meta-programming requires advanced programming skills. In this work we investigate three tasks in the context of scientific computing, where we utilize modern programming techniques as well as the Boost libraries. We introduce our approaches and show that applying those techniques and the Boost libraries results in highly versatile, maintainable, compact, and extendible code.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General;
D.2.13 [**Software Engineering**]: Reusable Software

Keywords

C++, Boost, Generic Programming, Meta-Programming, Functional Programming

1. INTRODUCTION

Modern programming techniques like generic and meta-programming are heavily utilized in the Boost libraries to provide versatile and extendible libraries to support C++ software developers [1]. One of the advantages of applying the Boost libraries is to make use of already available functionality, hence, reducing development time. For example, the Boost Spirit Library (BSL) [2] allows for the implementation of a versatile, extendible, and high-performance parser with minimum development effort. Additionally, the Boost libraries lower the entry barrier for utilizing modern programming techniques, like meta-programming. For example, the Boost Meta-Programming Library (MPL) [3] as well as the Boost Fusion Library (BFL) [4] provide algorithms and datastructures for the compile-time domain. Furthermore, the Boost libraries make upcoming C++ language features available, before they are formulated in the standard, hence, enabling the programmer to investigate possibilities on the frontier of the C++ programming language. The fact that various aspects of Boost have been adopted to the new C++11 standard underlines the importance and the influence of the Boost libraries to the overall C++ development [5].

In this work we focus on the application of modern C++ techniques and libraries to scientific software development. The scientific setting hugely benefits from open-source implementations, as available work can be accessed and extended, which ultimately reduces overall development time. However, modern programming techniques as well as Boost libraries, which heavily rely on those techniques, might be primarily only utilizable by enthusiasts or computer scientists. For example, tag-dispatching, concepts, and traits are typically not part of a C++ user's skill set.

However, the application of Boost libraries, and modern programming techniques in general, by engineering software tools has expanded over the recent years [6][7]. Generally, in an academic environment, especially in the context of engineering, the application driven focus on software development often supersedes the necessity to invest time and effort to implement software with a focus on extendibility, maintainability, and flexibility. In this work we depict three application scenarios of modern programming techniques and several Boost libraries in the context of scientific computing. We will show that the additional implementation effort results in highly extendible, maintainable, compact, and versatile code.

Generally, the tasks are related in the sense that they deal with a set of components in different contexts. Section 2 introduces an approach for a lightweight scheduler for a component framework. The Boost Graph Library (BGL) [8] as well as the Boost Phoenix Library (BPL) [9] are utilized to implement a plugin scheduler for sequential and parallel executions with minimum implementation effort. Section 3 discusses an advanced property-based selection method for a set of algorithms based on the BFL and the MPL. Section 4 depicts an approach to generalize geometrical algorithms by utilizing the generic paradigm. The theoretical background as well as implementations are discussed. Section 5 summarizes the presented topics.

2. LIGHTWEIGHT PLUGIN SCHEDULER

One way to introduce extendibility to a software project is to aim for an object-oriented framework approach [10][11]. Such an environment can be combined with a pluggable extension approach to support long lifetime support by increased maintainability and extendibility [12][13][14].

Our plugin scheduler is motivated from the area of semiconductor device simulation, yet applicable in a general setting. A plugin-based framework approach offers the required flexibility to support a heterogenous set of simulation tools as each tool is wrapped into a plugin and therefore is accessible by a unified interface. This interface can be utilized by the framework to drive and control the execution of the tool in a unified way. An important aspect, however, is, that the various plugins have different input data requirements, for example, distributions of various physical quantities. Those requirements have to be met before the plugin, and ultimately the simulation tool itself, can be executed, as the input data is vital for the internal simulation algorithm.

2.1 Sequential Execution Flow

Generally, a framework must offer a scheduler which ensures the proper execution of the set of supported plugins. This scheduler has to take the various input dependencies into account and, based on this, impose an ordering of the plugins with regard to their individual execution. Problems like this are typically referred to as dependence problems, which can be modeled as a graph. In scenarios, where tasks or jobs are associated with the vertices of a graph, the graphs are typically referred to as task graphs [15]. Graphs, where the nodes refer to tasks and one task cannot be executed before the preceding nodes are finished, are typically referred to as precedence graphs [16]. The BGL can be used to represent such graphs by a versatile and efficient datastructure. Furthermore, the BGL provides graph algorithms which can be utilized to implement a scheduler for the framework.

In the following, a straightforward approach for a basic scheduler based on the BGL is presented. It clearly shows that with basic BGL skills a scheduler for a plugin framework can be implemented with a minimum amount of development effort. We will show that the approach is flexible, maintainable, easy-to-use, and extendible.

First, a graph is defined which supports the problem at hand.

```
1 typedef boost::adjacency_list<
2    boost::vecS,
3    boost::vecS,
4    boost::directedS,
5    boost::property<boost::vertex_name_t,std::string>
6 > Graph;
7 Graph graph;
```

Listing 1: A graph datastructure is defined.

The graph datastructure `adjacency_list` is customized to reflect the specific requirements for the graph (Line 1). We use a `std::vector` (`boost::vecS`) datastructure, both for the vertex and the edge container, as it is not expected that many vertices and edges are added or removed during the execution (Lines 2,3). If that would not be the case, the `std::list` (`boost::listS`) datastructure can be used as this container supports fast insertions and deletions due to its doubly-linked list structure. However, this approach introduces memory overhead, which might be a concern for huge graphs. To map the execution flow, the graph is a directed graph (`boost::directedS`; Line 4). Additionally, the primary algorithm used on the graph, `topological_sort`, requires the graph to be a directed acyclic graph.

The term directed refers to the fact that an edge in the graph points from a source vertex to a sink vertex, whereas acyclic denotes that there must not be cycles in the graph. Finally, we associate string data (`boost::property<...>`) with each vertex (Line 5), which allows to store the plugin names on the graph vertices.

The next step is to build-up the graph by adding vertices and edges, also referred to as nodes and connections, respectively. The datastructure `graph` is populated with vertices by traversing the set of active plugins and adding exactly one node per plugin. In this way a node represents a plugin in the graph. During this process the dependencies of each plugin are stored for the subsequent step of connecting the vertices of the graph to reflect the input/output relations.

In the following code snippet the set of plugins is traversed and mapped to nodes in the graph. These nodes are then connected to finally form the graph.

```
1 for(plugin_iter = plugins.begin();
2     plugin_iter != plugins.end(); plugin_iter++) {
3   Graph::vertex_descriptor vertex =
4     boost::add_vertex((*plugin_iter)->name(), graph);
5   quantities[vertex] = (*plugin_iter)->output();
6 }
```

Listing 2: For each plugin a vertex is added to the graph datastructure.

A set of plugins (`plugins`) is traversed by an iterator (`plugin_iter`; Line 1,2). The identifier string is retrieved by accessing a member function `name()` of the dereferenced iterator and stored with the newly created graph vertex by utilizing the BGL provided insertion function (`boost::add_vertex`; Lines 3,4). Note that the elements of `plugins` are pointers to the different virtually overloaded plugin objects. The output data provided by the specific plugin is accessed by a member function `output()` and stored in an associative datastructure `quantities` as a value for a specific vertex `vertex` (Line 5).

At this point all the plugins are mapped to nodes in the graph datastructure. However, the graph has yet to be established by connecting the vertices related to their individual input/output dependencies:

```
1 for(plugin_iter = plugins.begin();
2     plugin_iter != plugins.end(); plugin_iter++) {
3   inputs = plugin_iter->input();
4   for(input_iter = inputs.begin();
5       input_iter != inputs.end(); input_iter++) {
6     /* find the node/plugin which provides the
7        required input  */
8     add_edge(source_id, sink_id, graph);   }}
```

Listing 3: The graph is created by connecting the vertices according to the input and output relations of the individual plugins.

A set of plugins is traversed by an iterator (Lines 1,2). Note that for each plugin to be connected in the graph the required input data set (`inputs`) is retrieved by a member function of the individual plugin (`input()`; Line 3). The set of inputs is traversed by using an iterator (`input_iter`; Lines 4-8). For each input traversed by the iterator, the vertex is determined which provides the requested data as an output (Lines 6,7).

When the vertex is found, a directed edge is added to the graph, which connects the determined source vertex with the sink vertex (Line 8). If no such vertex is found, the dependence is not met and an error indicating that the graph could not be resolved should be thrown. Figure 1 depicts an exemplary graph. The varying number of dependencies between the various plugins represent different quantities. Further note that a plugin might have more than one input.

In graph theory the number of input and output edges is typically referred to as in-degree and out-degree, respectively. Generally, the degree of a vertex is understood to be the number of incident edges. Applied to our problem at hand, the nodes of the graph have non-constant degree, as the plugins might have a different number of input and output edges. Therefore, the scheduler has to be capable of dealing with varying numbers of inputs and outputs of each individual plugin.

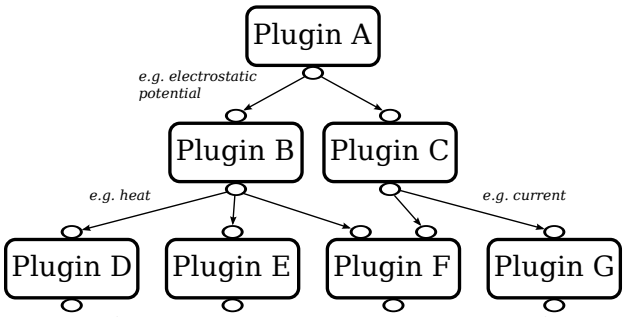

Figure 1: A plugin dependence graph is shown. A plugin can have more than one input or output.

At this point the graph is set up and can be analyzed regarding the execution order by the so-called topological sort algorithm [17]. The BGL provides an implementation of this algorithm, which yields the required execution order. This ensures that the plugins are only executed, when the individual inputs are available. Note that this fact models the definition of a precedence graph. The algorithm yields the scheduled sequence of vertices. As those vertices are placeholders for the actual plugins, the scheduler has to execute the plugins in exactly the same order as computed by the topological sort algorithm.

2.2 Parallel Execution Flow

At this point the presented straightforward approach deals only with a sequential execution flow, meaning that a plugin is executed one after another on a single computing unit, for example, a CPU-core of a workstation. This situation changes in the context of a parallel execution environment, for which the presented graph approach can be extended to reflect the requirements of task scheduling for a system of multiple computing units [18]. Several approaches based on graphs and on executing different algorithms are available, for example, depth-first search and breadth-first search. In the following an approach is depicted to implement a basic scheduler for scheduling a task graph for a parallel environment. The approach is based on the so-called list scheduling technique [16]. The underlying principle is to set up a list of prioritized tasks. Each task is checked whether or not it has already been scheduled and if the incident parent tasks are finished. The list is repeatedly processed until all of the tasks are scheduled.

Generally, different approaches are available to prioritize the tasks of a task graph. One of the straightforward approaches is to use a topologically ordered sequence. Therefore, the presented sequential execution flow approach can be directly extended to generate the priority list. In comparison to the scheduling approach for a single execution flow, the graph type needs to be changed from a directed graph (`directedS`) to a bidirectional graph (`bidirectionalS`) to provide access to the parent vertices for a given vertex via the `in_edges()` function.

The BGL uses integer values to uniquely identify and represent vertices. This introduces the need for a mapping from the integer-based vertices to the actual plugins. Such a mapping conveniently allows for the evaluation of the scheduling based on the graph datastructure and to relate the results to the actual plugin objects. An exemplary approach to set up such an associative relation is depicted in the following. Note that this associative datastructure is also utilized in Section 2.1.

```
1 typedef std::map<std::size_t,
2    boost::shared_ptr<plugin_base> >       PluginMap;
3 PluginMap pluginmap;
4 // define a set of plugin smart pointers
5 boost::shared_ptr<plugin_base> pa(new plugin_a);
6 boost::shared_ptr<plugin_base> pb(new plugin_b);
7 // map the plugins with a specific vertex index
8 pluginmap[0] = pa;
9 pluginmap[1] = pb;
```

Listing 4: A mapping from the graph vertices to the actual plugins is provided.

The associative container is declared with the unsigned integral key type `std::size_t`, as the indices are only positive integers. We base our investigations on a straightforward plugin class hierarchy with virtual function overloads, as this setup reflects the run-time nature of the stated problem. Therefore, the `PluginMap` datastructure holds pointers to the base class `plugin_base` of the run-time class hierarchy as a value type (Lines 1,2). Plugin smart-pointers are created (e.g. `plugin_a`; Lines 5,6) and associated with a specific vertex index from the present task graph (e.g. `pluignmap[0]=pa`; Lines 8,9). The priority list is generated in the same way as the schedule for the sequential execution flow by utilizing the topological sort algorithm.

The required steps of checking whether or not a task has already been scheduled and the parent tasks have finished the execution can be implemented differently. We aim for a compact implementation, meaning that with a minimum amount of user-level code it should be possible to program an intricate system. We refer the term user-level to the part of the implementation which is very likely to undergo maintenance and expansion on a regular basis.

Our approach is to utilize the BPL. This library enables functional programming in C++, which can result in highly readable and modular user-level code and therefore significantly increases the maintainability and ultimately the extendibility of the implementation [19]. Furthermore, we will see, that due to this approach the required lines of code to implement the whole algorithm are reduced significantly. An in-depth view on our approach is provided in the following.

At first, the core part of the final user-level expression of the scheduling algorithm is depicted to present the goal of the whole implementation approach.

```
1 std::for_each(prioritized.begin(), prioritized.end(),
2    if_(!is_scheduled && is_executable) [ execute ] );
```

Listing 5: A one-line implementation of the list scheduling approach based on the BPL is depicted.

Due to the application of the functional programming paradigm, more specifically the BPL, the core part of the scheduler is minimized to a single line of code. The sequence of prioritized plugins (`prioritized`), computed by the topological sort algorithm, is traversed by utilizing the `for_each` traversal (Line 1). The third parameter of the traversal is utilized to insert a Boost Phoenix functional expression (`if_[..]`; Line 2). This expression generates a functor and is therefore utilizable by the standard traversal algorithm.

In this case the functional expression is a so-called lazy-statement condition. As an argument for the condition we utilize two algorithms, named `is_scheduled` and `is_executable`, which model the Boost Phoenix actor concept (Line 2). These algorithms implement the logic for the discussed list scheduling technique. If the condition is met, the individual plugin is enqueued for execution, which, for example, could mean that the plugin is assigned to the next processor with the earliest start-time.

For the sake of simplicity, we do not focus on the implementation of the `execute` actor. The difference primarily is, that the executor does not have to access the Boost Graph datastructure, as its only task is to retrieve the actual plugin object and enqueue it for execution. It is important to note that the presented code snippet in Listing 5 needs to be executed repeatedly for the prioritized list, until all tasks have been scheduled, as the plugins might have to wait until the input data is available and, therefore, can be executed.

Although the above code snippet is very concise, it requires the implementation of the applied algorithms within the conditional functional expression. In the following, the implementation of one of the presented Boost Phoenix actors is presented, namely the `is_executable` actor. The implementations of the other two actors `is_scheduled` and `execute` is analogous in regard to the Boost Phoenix specific implementation details. Generally, the implementation models the Boost Phoenix actor concept. For the sake of simplicity, only the implementation of the functor part is depicted in Listing 6.

A nested `result_type` member-type is provided which holds the return type of the functor (Lines 2, 7). The result type implementation of the functor models the `result_of` concept [20]. Two state objects are retrieved from the Boost Proto expressions provided by the functors argument list, the graph and the associative plugin-id container (Lines 8, 9, 11, 12). The actor's arguments are collected in a so-called environment which itself is part of the context (Line 10). The `plugin_id` object is accessed which has been passed as an argument to the actor (Line 14). This plugin identifier, which represents a Boost Graph node, is used to access the actual plugin object which finally should be tested whether it can be queued for execution.

Note that for the sake of simplicity we skipped the Boost Phoenix specific implementation details of retrieving the state objects and the actors arguments. At this point all the required data is available and the parent plugins can be determined by utilizing BGL algorithms. The set of incoming edges, computed by the `in_edges()` function, is traversed by an edge iterator `ei` (Lines 16-23). For each edge the source vertex is determined by the `source()` function (Line 20). The source vertex is mapped to the actual plugin object by utilizing the `find()` member-function of the plugin map `plugins`. The execution state of the parent plugin is accessed by utilizing the implementation specific `state()` member-function. The state is tested whether the parent is finished. If only one of the parents is not finished, the plugin under test cannot be executed. Only when all of the parent plugins have finished their execution, the plugin can commence execution.

```
1  struct executable_eval {
2     typedef bool result_type;
3
4     template <typename GraphExpr,
5               typename PluginMapExpr,
6               typename Context>
7     result_type
8     operator()(GraphExpr     const& graph_expr,
9                PluginMapExpr const& plugins_expr,
10               Context            & ctx)    const {
11       /* retrieve the graph and plugins objects
12          from the state expressions */
13
14       /* retrieve the plugin id from the context */
15
16       typename graph_traits<
17          Graph>::in_edge_iterator       ei,edge_end;
18       for(tie(ei,edge_end)=in_edges(plugin_id,graph);
19           ei != edge_end; ++ei) {
20          if( (plugins.find(source(*ei,graph))
21              ->second)->state() != FINISHED )
22             return false;
23       }
24       return true; } };
```

Listing 6: An implementation of an algorithm modeling the Boost Phoenix actor concept is shown.

2.3 Conclusion

The discussed implementation for the sequential flow handles serial execution of a sequence of plugins only. However, the evaluation of the execution orders requires about 30-50 lines of code, which can be considered to be highly maintainable. Regarding the approach for the parallel execution flow, the sequential implementation can be directly extended to support scheduling of a task graph for a parallel computing environment. The presented functional approach has the benefit of a highly concise formulation of the algorithm. However, this approach requires some additional code in regard to the algorithm implementations. Approximately 200-300 lines of code are necessary in addition to the sequential implementation to implement the parallel version. It is important to emphasize, however, that only a couple of code lines are actual user-level code. Generally, the majority of the implementation resides in the BGL. This ultimately reduces the maintenance efforts to a minimum. Furthermore, as the BGL is a mature library, interface changes in upcoming versions are not to be expected. The utilized topological sort algorithm is expected to perform well even for large sets of vertices and plugins respectively, as the algorithm has a time complexity of $O(V+E)$, with V and E being the number of vertices and edges in the graph.

In regard to the goal of high flexibility and extendibility for the scheduler, the BGL does not only provide a significant set of graph algorithms, but also a sophisticated generic datastructure which allows to assign weights on nodes and edges. The ability to associate weights on vertices and edges enables to model even more complex task-graph schedulers for parallel environments, for example, to model the network and processing capabilities of different computing nodes.

3. META PROPERTY-BASED SELECTION

The previous section discussed the scheduling of a set of components. Another task is to select a subset of those components based on certain properties during compile-time. In this section we give a motivation and depict an application scenario for such a meta-selection.

Scientific simulations typically deal with sets of algorithms as, for example, different geometrical algorithms are available for different geometrical entities. Eventually, a specific algorithm has to be selected from a set of algorithms based on certain properties. In a run-time based environment this decision is naturally made during execution time. However, if the information for the decision process is already available during compile-time, a run-time selection results in unnecessary overhead, which introduces the need for a compile-time selection mechanism. This section introduces an approach based on the BFL and the MPL. First the algorithm is discussed and second, an implementation is provided.

3.1 Adding Meta-Selection Capabilities

In its essence the selection algorithm determines the subset of available tools which fulfill the requested properties during compile-time (Figure 2). The following discussion presents a compile-time property-based selection approach applied to the field of mesh generation [21]. This research field provides a variety of algorithms and publicly available tools, which eventually introduces the need for a concise interface for the individual implementations and, consequently, a generic approach to select specific versions based on properties [22]. The concrete goal is to design a mechanism which allows for the selection of a mesh generation tool based on certain properties during compile-time. For example, the mesh generation tools provide varying support for the dimensions of the input geometry, e.g., two-/three-dimensional mesh generation tools.

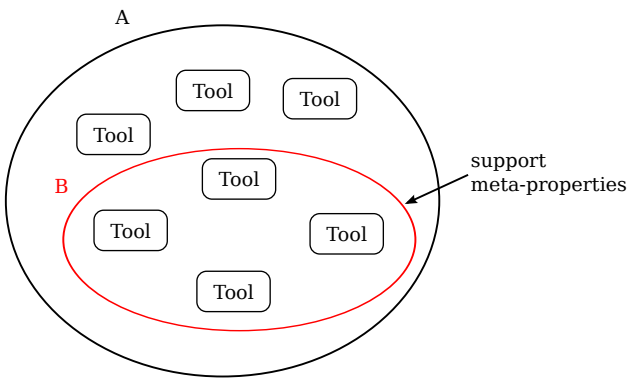

Figure 2: The compile-time selection algorithm selects the subset $B \subseteq A$ of elements supporting the requested properties.

Another example is the utilization of different underlying mesh generation algorithms, like the Advancing Front algorithm or the Incremental Delaunay algorithm [21]. Although we base our discussion on the field of mesh generation, the approach can be directly applied to different fields, like computational geometry, where, for example, a set of different algorithms can be maintained based on the algorithm specific properties, e.g., high-performance or robust specializations. In general, the presented approach extends the functionality of the `filter_view` algorithm which checks against a single predicate [3][4]. Our approach, however, allows for varying numbers of properties both for the provided and for the requested set of properties. For example, an algorithm provides two properties, being two-dimensional mesh generation and simplex mesh elements. If the requested properties contain only the type of mesh elements, this algorithm is still being chosen although only one of the available properties is requested.

The basis for this approach is to attach a property datastructure to each element of the algorithm set, which in our example refers to the set of available meshing tools. Note that this set refers to the superset A in Figure 2. The attached datastructure should hold information about the individual properties of the algorithm, for example, the type of the generated mesh elements. Generally, this datastructure should be flexible, extendible, and support compile-time access. Here, flexibility and extendibility refers to the fact that properties of various types and of arbitrary number should be supported. The associative, heterogenous `map` container provided by the BFL copes with the introduced requirements. Note that heterogeneity refers to the fact that objects of arbitrary datatype can be stored. The following code snippet depicts an exemplary implementation of a mesh generator wrapper class with such an embedded, associative property container. This property container is later on accessed by our compile-time algorithm to investigate the suitability for a given set of properties.

```
1 struct mesh_generator_one {
2  // ...
3  typedef result_of::make_map<
4     dimg, dimt, cell,
5     three,three,simplex>::type  properties_type; };
```

Listing 7: An associative property datastructure is embedded in an available algorithm implementation.

`mesh_generator_one` refers to an exemplary implementation of a mesh generator wrapper class. Such a wrapper can be used to map a specific interface of a third-party mesh generator library to a unified interface. The nested `properties_type` can be identified as a concept requirement, as its presence is expected later on by our compile-time facilities [23]. The keys (Line 4) map to specific values (Line 5). `dimg` and `dimt` refer to the dimension of the geometry and topology space of the generated mesh elements, respectively. The `cell` key and its value `simplex` refer to the specific type of a simplicial mesh element type [24]. The presented example indicates, that the mesh generator supports the generation of a three-dimensional volume mesh based on tetrahedral mesh elements. New property entries can be conveniently added, if required. The used tags in Line 4,5 are empty `structs`, for example, `struct dimg{};`. This approach has a high degree of flexibility, due to the support of properties of arbitrary type.

However, the approach of embedding the properties datastructure in the algorithm implementation is intrusive, meaning that existing code has to be altered. A more generic approach is to provide a decoupled, tag-dispatched meta-function mechanism which derives for a given tag (which relates to a specific algorithm) the property datastructure. A meta-function is a class or a class template which provides a nested `type` typedef [3]. Therefore, a meta-function can be interpreted as a compile-time variant of a common function, which computes for a set of input parameters, the type template parameters, an output, which is accessible via the nested `type`. The following code snippet depicts a possible implementation for this approach.

```
1 namespace result_of {
2 template<typename T>
3 struct properties{typedef error type;};
4
5 template<>
6 struct properties <mesh_generator_one> {
7  typedef typename result_of::make_map<
8    dimg, dimt, cell,
9    three,three,simplex,
10  >::type    type; };
11 }
```

Listing 8: A tag-dispatched meta-function mechanism provides non-intrusive generation of the properties container.

We embed the meta-function facility in the `result_of` namespace, to follow the coding style of, for example, the BFL (Line 1). The default meta-function specialization evaluates to an error indicating type, e.g., `struct error{};`(Lines 2,3). An arbitrary number of specializations can be implemented based on unique tags (Lines 5-10). This tag-dispatched meta-function can then be used to generate the actual property container for a specific mesh generation implementation, similar to the implementation depicted in Listing 7.

```
1 typedef result_of::properties<
2    mesh_generator_one>::type  properties_type;
```

Listing 9: A non-intrusive, generic approach assigns a property container to a specific class by utilizing the tag-dispatching technique.

At this point, the elements of the tool set offer a container which provides information about their properties. The next step is to implement a mechanism which, for a given input property container, traverses the elements of the algorithm set and determines whether the individual elements support the required properties. The result of this operation is a container of feasible algorithms.

In the following, the core parts of our approach are presented. At first, the user-level code is shown and then an in-depth discussion of the underlying components is provided. The following code snippet utilizes the facility to compute the set of tools which support a set of properties out of a set of available tools.

```
1 typedef typename filter_fold::apply<
2   Properties, AvailableTools >::type   ResultTools;
```

Listing 10: The compile-time facility is utilized to compute the subset of tools which support the required properties.

The class `filter_fold` models the MPL concept of a meta-function class. A meta-function class is supposed to provide the nested meta-function `apply`.

This meta-function is utilized to compute the actual set of the input set `AvailableTools`, which supports the set of properties `Properties`. Note that the container `AvailableTools` and `Properties` are modeled by a MPL `vector` and a BFL `map`, respectively.

In the following, the internals of the meta-function class `filter_fold` are presented. Note that the implementation is inspired by the `do_the_bind` example of the BFL.

```
1 struct filter_fold {
2   struct fold_op {
3     template <typename Sig>    struct result;
4
5     template <class S, class ToolSet, class Property>
6     struct result< S(ToolSet &,Property &) > {
7       typedef typename mpl::filter_view<
8         ToolSet, check<Property>
9       >::type  type; }; };
10
11  template <typename Properties, typename ToolSet>
12  struct apply : fusion::result_of::fold<
13    Properties, ToolSet, fold_op>::type { }; };
```

Listing 11: The implementation of the meta-selection algorithm is based on the `filter_view` and the `fold` meta-functions.

The class `filter_fold` is based on two nested classes: First, the test operation `fold_op` is implemented as a class which offers a nested `result` meta-function (Lines 2-9). The implementation models the `result_of` concept which is part of the new C++11 standard [5][20]. This nested meta-function utilizes the `filter_view` algorithm provided by the MPL. This filter algorithm computes the subset of `ToolSet`, the elements of which satisfy a specific property under test (`check<Property>`; Lines 7,8). `fold_op` is executed once for each `Property` of the property set `Properties`. Second, a binary meta-function `apply` is provided (Lines 11-13). This meta-function invokes the `fold` sequence traversal algorithm provided by the BFL based on the set of properties (`Seq`), the set of tools (`State`), and the test operation which checks the feasibility of each tool (`fold_op`).

The following code snippet introduces the final internal part, being the test class `check<...>` which enables the `filter_view` algorithm to verify if a certain property is supported by a specific tool. The implementation models the unary meta-function class concept. For the sake of simplicity only the meta-function implementation of the nested `apply` structure is discussed.

```
1 typedef typename result_of::value_of<
2   typename result_of::find_if<
3     typename result_of::properties<Tool>::type,
4     is_same<mpl::_,PropertyPair>
5   >::type
6 >::type                        find_result_type;
7 typedef typename is_same<
8   PropertyPair, find_result_type>::type   type;
```

Listing 12: A test meta-function class is provided which is used by the `filter_view` meta-function. The implementation evaluates whether or not a property is supported by a specific tool.

The algorithm works as follows: First, the `find_if` algorithm is utilized to determine whether the tool under consideration (`Tool`) provides the property under test (`PropertyPair`; Lines 2-5).

The supported properties of the tool are retrieved by utilizing the `properties<>::type` meta-function introduced in Listing 9 (Line 3). The property `PropertyPair` is actually a BFL `pair`, as the fold operation (Listing 11) operates on the elements of the associative properties map, which are pairs. Second, the actual type stored at the position of the returned iterator of the `find_if` algorithm is determined by the `value_of` algorithm. Third, the Boost Type-Traits Library (BTTL) [25] is utilized, more specifically, the `is_same<>::type` meta-function is applied to test whether the result of the find-operation is the same as the requested property (Lines 7,8). A MPL boolean is returned indicating the result of this evaluation.

3.2 Conclusion

The presented meta-selection facility requires around 30-50 lines of code with additional 3-5 lines of code for each attached property container to the individual tool or algorithm specializations. Due to the utilization of the MPL, the BFL, and the BTTL the implementation effort is highly reduced. The major advantage of the presented approach is the high degree of flexibility and extendibility. The selection process can be applied to arbitrary scenarios, especially due to the discussed non-intrusive approach to relate properties to existing implementations. Furthermore, the utilization of the associative BFL container supports not only arbitrary keys and values for the properties, but also the number of properties can vary.

4. ALGORITHM GENERALIZATION

So far we discussed scheduling and filtering of a set of components. For the sake of extendibility and versatility it is important to generalize those components to maximize the degree of utilization, hence, minimizing long-term development efforts. This section investigates an approach for generalizing geometrical algorithms. One of the core aspects of generic programming is the notion of abstraction [26]. Abstraction, also referred to as generalization, enables to apply implementations in a variety of situations. A typical example would be from the field of computational geometry, where the algorithms may be implemented generalized in regard to the dimension of the geometry space, e.g., the distance algorithm which computes the distance between two point-vectors of arbitrary dimension [27]. We will in particular discuss a couple of geometrical algorithms and derive an approach to generalize them. We first analyze the algorithms, group them, and lift them, which ultimately reveals the underlying generalized algorithm. A possible implementation approach is finally depicted, which directly utilizes the discussed generalization approach.

4.1 Background

The basis for our generalization approach is based on investigating the algorithms not only with respect to geometry but also in regard to topology. Informally, geometry deals with shape, size, and position, whereas topology is about continuity and connectivity [28]. Generally, a geometrical algorithm implicitly contains not only geometrical information, like, a geometrical space $\mathbb{R}^d$, but also topological information, like, the number of vertices of a polygon on which the algorithm is evaluated on. Table 1 depicts the geometrical and topological informations of different algorithms in examplary settings.

	Geometry	**Topology**
Line Length, $\mathbb{R}^3$	3D	1D,`S`/`C`
Triangle Area, $\mathbb{R}^2$	2D	2D,`S`
Tetrahedron Volume, $\mathbb{R}^3$	3D	3D,`S`
Cube Volume, $\mathbb{R}^3$	3D	3D,`C`

Table 1: The geometrical and topological informations of different algorithms are depicted. D denotes the dimension of the respective space, `S` and `C` refer to simplex and cube topology, respectively. Simplex and cube can be informally interpreted as the topological base type of the geometrical entity which the algorithm processes. Note that the algorithms can also be embedded in different geometrical spaces.

The depicted extraction of information in Table 1 is the basis for the algorithm generalization. An important step towards generalization by extracting the geometrical entity of an algorithm is to map geometrical entities to topological ones by introducing the notion of a cell, and more specifically a k-cell where k denotes the topology dimension (Table 2) [24].

k-**cell**	**topological object**	**geometrical object**
0-cell	vertex	point
1-cell	edge	line
2-cell	face	triangle, quadrilateral, ...
3-cell	cell	tetrahedron, cuboid, ...

Table 2: The relations between an arbitrary k-cell (**left**), topological objects (**middle**) and the geometrical counterparts (**right**). A unique mapping from a k-cell to a geometrical entity can only be realized for dimensions of up to one. For $k > 1$ the mapping to geometrical entities is not-unique.

There exists only a unique relation between the geometrical entities and the topological counterparts for $k = 0, 1$. A unique relation for $k > 1$ can only be achieved by additional information called cell topology (Table 3). The cell topology has already been introduced in Table 1 as the so-called base type.

k-**cell**	**Cell Topology**	**Geometrical Entity**
0-cell	simplex/cube	point
1-cell	simplex/cube	line
2-cell	simplex	triangle
2-cell	cube	quadrilateral
3-cell	simplex	tetrahedron
3-cell	cube	cuboid

Table 3: A unique mapping from the topological cell objects to the geometrical entities can only be introduced by combining the information of the dimension, k, with the cell topology, e.g., simplex. For $k = 0, 1$ the cell topology is obsolete, as it always maps to a point and line, respectively.

At this point we can conclude that to uniquely map a geometrical entity, like a triangle, to a topological object, like a cell, a topology dimension and a cell topology is required (Figure 3).

With the introduced notion of a topology, and a k-cell especially, the geometrical algorithms can be investigated regarding generalization.

We base our investigations on the following algorithms:

- length of a line
- area of a triangle
- volume of a tetrahedron
- point in triangle test
- point in tetrahedron test

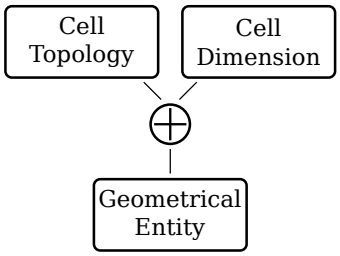

Figure 3: Combining cell topology and cell dimension yields a unique mapping to a geometrical entity.

The first step is to group the set of algorithms according to their underlying behavior (Table 4).

algorithm	generalized algorithm
length of a line	Metric quantity
area of a triangle	Metric quantity
volume of a tetrahedron	Metric quantity
point in triangle test	k-cell in q-cell
point in tetrahedron test	k-cell in q-cell

Table 4: The relations between the algorithms and their respective generalized versions are depicted. **Top:** The term *metric quantity* is introduced to refer to the category of measuring algorithms. **Bottom:** The k-cell notation is used to generalize the inclusion tests. Note that the generalized versions lack any dimensional information as well as the type of geometrical entity which is processed.

Finally, analyzing the grouping of the geometrical algorithms in Table 4, the view in Figure 3 can be extended to ultimately reveal an approach to generalize geometrical algorithms (Figure 4).

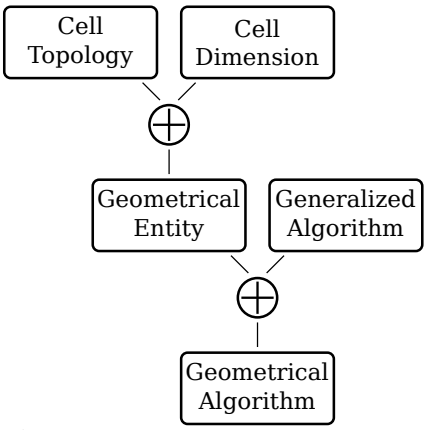

Figure 4: A generalized algorithm can be formulated by extracting the geometrical entity of a geometrical algorithm.

4.2 A Generic Algorithm Interface

The presented generalization approach can be directly implemented by utilizing the template specialization technique. This straightforward approach can be used to specialize for various cell dimensions and topologies. The following code snippets depict our approach based on the metric quantity generalization. A default base class is provided which will be specialized based on the cell dimension and the cell topology.

```
1 template < int Dimension, typename Topology >
2 struct metric_quantity { };
```

Listing 13: A default base class is shown which will be specialized based on the k-cell dimension and topology.

Partial template specialization for one-dimensional entities is shown, where a 1-cell maps uniquely to a line, regardless of the topology (Table 2).

```
1  template < typename Topology >
2  struct metric_quantity < 1, Topology > {
3     template<class> struct result;
4
5     template<class F, typename Cell>
6     struct result<F(Cell)> {
7         typedef double type;
8     };
9
10   template < typename Cell >
11   typename metric_quantity::result<
12     metric_quantity(Cell)>::type
13   operator()(Cell& cell) const {
14     return boost::geometry::distance(cell[0],cell[1]);
15   }};
```

Listing 14: The distance function is utilized for cell objects of dimension one, regardless of the topology.

The `result_of` concept is modeled by nesting the metafunction `result` in the functor implementation [20] (Lines 3-8,11-12). This approach is important as it provides a `Cell`-type dependent return type evaluation, which is vital as the return type might change for different cell-types. Each specialization contains the actual algorithm, for example, the BGL based `distance()` function (Lines 10-15) [27]. In the presented example, we expect the `cell` object to provide access to the individual BGL point vector objects by the `[]`-operator overload (Lines 13,14).

Another specialization is the area of a triangle, which is presented in the following.

```
1  template < >
2  struct metric_quantity < 2, tag::simplex > {
3     template<class> struct result;
4
5     template<class F, typename Cell>
6     struct result<F(Cell)> {
7         typedef double type;
8     };
9
10   template < typename Cell >
11   typename result< Cell >::type
12   operator()(Cell& cell) const {
13     return boost::geometry::area(cell);
14   }};
```

Listing 15: The area of a triangle is computed for a 2-cell with simplex topology.

A 2-cell with `simplex` topology uniquely maps to a triangle. The functor utilizes the `area` algorithm of the BGL (Line 13).

For the sake of simplicity, the cell object is expected to model the concept of a BGL polygon, hence, the `area` algorithm can be directly applied to the cell object. However, a more efficient approach could be to utilize a dedicated triangle area algorithm at this point, as the BGL's `area` function aims at processing, for example, run-time based polygon objects.

Figure 5 depicts the mapping of the presented generalization approach shown in Figure 4 to the discussed tag-dispatched implementation approach.

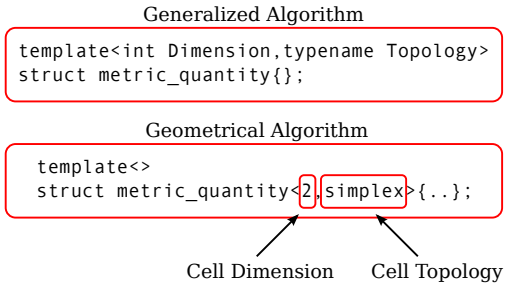

Figure 5: The various parts of the discussed generalization approach are identified in the tag-dispatched implementation. **Top:** The base class `metric_quantity` represents the generalized algorithm, which by itself contains no program logic. **Bottom:** Only with the k-cell topology and dimension the generalized algorithm concretely maps to a geometrical one.

It is important to note that the discussion so far has only dealt with compile-time information. This, however, is insufficient for supporting typical run-time entities, like, polygons. In such a case, the k-cell topology and dimension is obviously only available during run-time, hence, introduces the need for a run-time based dispatch. The concept of the dispatch follows the introduced compile-time approach, however, for the sake of simplicity we focus on the compile-time case.

4.3 Utilization by a Generic Datastructure

Our implementation follows one of the key aspects of the generic paradigm, being the separation of algorithms from datastructures. The presented tag-dispatched implementation represents the algorithm. This algorithm can be utilized in conjunction with, for example, a mesh datastructure which supports compile-time handling. Compile-time handling refers to the ability to access datastructure relevant information during compile-time, for example, the number of incident cell vertices. In the following we discuss an application scenario where our presented generic algorithm is used in conjunction with a generic compile-time mesh datastructure provided by the ViennaGrid library [29].

First, a domain object is created which holds the actual mesh information, like the point-vectors and the mesh elements.

```
1 typedef  config::triangular_2d              Config;
2 typedef  result_of::domain<Config>::type    Domain;
3 Domain domain;
4 domain.push_back(Point(0,0));
5 ..
6 domain.push_back(Cell(...));
7 ..
```

Listing 16: A ViennaGrid domain is created.

The domain is configured during compile-time (Line 1,2). Various datastructure configurations are supported, like a three-dimensional cuboid mesh. The domain is instantiated (Line 3) and loaded with geometry and topology information (Lines 4-7).

At this point the datastructure is populated and our previously introduced generic algorithm interface is utilized for computations. For example, a typical task might be to compute the area of each mesh element of a two-dimensional mesh. This can be realized by traversing the set of mesh elements and utilizing the generic metric quantity algorithm for each of them, like depicted in the following.

```
1 typedef boost::result_of<
2    metric_quantity_gen(Cell) >::type quan_type;
3 ..
4 CellRange cells = ncells(domain);
5 for(CellIterator cit = cells.begin();
6     cit != cells.end();++cit) {
7    quan_type quan = compute_mq(*cit);
8 }
```

Listing 17: The metric quantities of all cells of a domain are computed.

We introduced an additional functor (`metric_quantity_gen`) on top of the tag-dispatched implementation introduced in Section 4.2. This functor extracts the required tags from the cell object type `Cell` and forwards them to the respective specializations. `metric_quantity_gen` models the `result_of` concept and as such provides return type computation (Lines 1,2). The set of mesh elements (`cells`) is extracted from the domain (Line 4) and traversed by an iterator `cit` (Line 5-8). The previously instantiated generic metric-quantity functor `compute_mq` is called for each mesh element (Line 7).

It is important to note that the presented implementation of computing the metric quantities is decoupled from the domain type. If, for example, the domain configuration in Listing 16 is changed from `triangular_2d` to `tetrahedral_3d` the volume of the tetrahedral mesh elements are computed. The algorithm adapts automatically during compile-time by analyzing the provided cell type. This generic approach allows to build up intricate implementations for various application scenarios, without actually changing the code base. Additionally, the application of the `result_of` concept allows for algorithm-specific return types. This might be of significant interest in the case of, for example, numerical robustness, where different high-precision datatypes can be used to achieve highly accurate results. Those facts combined outline the generic implementation style of our approach.

4.4 Conclusion

Our approach to generalize geometrical algorithms based on the notion of topology, and of a k-cell especially, can be directly utilized for a generic algorithm interface. We have shown that our generic tag-dispatched algorithm hierarchy maps directly to our generalization approach. Additionally, our implementation works best in conjunction with datastructures which support compile-time handling, as the tag-dispatching facilities can be attached with the datastructures meta-system, ultimately nullifying run-time overhead for algorithm dispatches.

5. SUMMARY

We have introduced three different implementation tasks from the field of scientific computing in detail. By applying modern programming techniques as well as the Boost libraries we achieve highly versatile, maintainable, and extendible code primarily due to the application of the generic programming paradigm. Therefore, our applications show that the additional time spent in learning advanced C++ skills pays off in the long run. Furthermore, our approaches can be used for further investigations, for example, in the context of a generic computational geometry library.

Acknowledgments

This work has been supported by the European Research Council through the grant #247056 MOSILSPIN. Karl Rupp gratefully acknowledges support by the Graduate School PDETech at the TU Wien.

6. REFERENCES

[1] The Boost C++ Libraries. http://www.boost.org/.

[2] The Boost Spirit Library. http://www.boost.org/libs/spirit/.

[3] The Boost Metaprogramming Library. http://www.boost.org/libs/mpl/.

[4] The Boost Fusion Library. http://www.boost.org/libs/fusion/.

[5] Matt Austern. Draft Technical Report on C++ Library Extensions. *ISO/IEC JTC1/SC22/WG21*, N1836, 2005.

[6] Karsten Ahnert et al. Odeint - Solving Ordinary Differential Equations in C++. *AIP Conference Proceedings*, 1389(1):1586–1589, 2011.

[7] Peter Gottschling et al. Integrating Object-Oriented and Generic Programming Paradigms in Real-World Software Environments. In *POOSC Workshop at ECOOP*, 2008.

[8] The Boost Graph Library. http://www.boost.org/libs/graph/.

[9] The Boost Phoenix Library. http://www.boost.org/libs/phoenix/.

[10] Greg Butler et al. Documenting Frameworks to Assist Application Developers. In *Object-Oriented Application*. John Wiley and Sons, 1997.

[11] Garry Froehlich et al. Designing Object-Oriented Frameworks. In *Handbook of Object Technology*, pages 1–30. CRC Press, 1998.

[12] Timothy R. Culp. Industrial Strength Pluggable Factories. *C++ Report*, 11(9), 1999.

[13] Dia Kharrat et al. Self-Registering Plug-ins: An Architecture for Extensible Software. In *CCECE*, pages 1324–1327, 2005.

[14] Tiago Quintino. *A Component Environment for High-Performance Scientific Computing*. PhD thesis, Katholieke Universiteit Leuven, 2008.

[15] Kunal Agrawal et al. Executing Task Graphs Using Work-Stealing. In *IPDPS*, pages 1–12, 2010.

[16] Yu-Kwong Kwok et al. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4), 1999.

[17] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009. ISBN 0262033844.

[18] Tracy D. Braun et al. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61(6), 2001.

[19] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2), 1989.

[20] The Boost Utility Library. http://www.boost.org/libs/utility/.

[21] Jonathan R. Shewchuk. Unstructured Mesh Generation. In *Combinatorial Scientific Computing*, pages 259–297. CRC Press, 2012. ISBN 1439827354.

[22] Josef Weinbub et al. High-Quality Mesh Generation Based on Orthogonal Software Modules. In *SISPAD*, pages 139–142, 2011.

[23] Gabriel Dos Reis et al. Specifying C++ Concepts. *SIGPLAN Notices*, 41(1), 2006.

[24] Guntram Berti. *Generic Software Components for Scientific Computing*. Dissertation, Technische Universität Cottbus, 2000.

[25] The Boost Type-Traits library. http://www.boost.org/libs/type_traits/.

[26] Gabriel Dos Reis et al. What is Generic Programming? In *LCSD*, 2005.

[27] The Boost Geometry Library. http://www.boost.org/libs/geometry/.

[28] Afra J. Zomorodian. *Topology for Computing*. Cambridge University Press, 2005. ISBN 0521836662.

[29] ViennaGrid. http://viennagrid.sourceforge.net/.