

Cross-Platform Development of High Performance Applications Using Generic Programming

Wolfgang Blochinger and Wolfgang Küchlin
University of Tübingen
Wilhelm Schickard-Institute
Symbolic Computation Group
Sand 14, D-72076 Tübingen, Germany
email: {blochinger,kuechlin}@informatik.uni-tuebingen.de

ABSTRACT

We investigate methods for creating highly portable parallel and distributed applications using the programming language C++. The paper elaborates on software engineering issues for efficiently mapping abstract class interfaces to system functionality — typically designed by applying the wrapper facade design pattern — on a large number of target platforms. We introduce a novel mapping technique which is based on generic programming. The paper presents a detailed comparison of the proposed method with two other commonly applied techniques, concluding that our approach represents the most suitable technique for creating highly portable C++ programs in terms of efficiency, ease of portability, and enforcement of correctness.

KEY WORDS

Software Development, Heterogeneous High Performance Computing, Generic Programming, Design Patterns

1 Introduction

With the advent of the grid computing [1] era, we must deal with a high degree of heterogeneity even in the field of high performance computing. From a system programmer's point of view, heterogeneous high performance computing mainly comprises the aspects of efficient interoperability, and of high portability of applications. The latter aspect is the focus of our paper.

This research was carried out in the context of the design and implementation of the Distributed Object-Oriented Threads System (DOTS) [4]. DOTS is a parallel programming toolkit for C++ that integrates a wide range of different computing platforms into a homogeneous system environment; e.g., it runs on realtime micro-kernels, Microsoft Windows, many flavors of UNIX, and on IBM Mainframes under OS/390 [5, 6]. The problem for DOTS and other software which must run on a variety of system platforms is to cope efficiently, and with little programming effort, with variabilities in the underlying operating systems.

Conceptually, operating systems of different vendors

are very similar. They provide the same kinds of abstractions for system functionality and resources, e.g. processes, threads, and sockets. However, their application programming interfaces (APIs) for accessing system functionality differ considerably. For example, in the Microsoft Windows API, a new program is started using the `CreateProcess()` system call. In UNIX systems this task is carried out by two different system calls: `fork()` creates a new process (a copy of the parent process), and `exec()` enables this process to execute a new program. This fact means that applications based on a specific API are not portable to other operating systems without awkward code modifications that spread through the entire program.

This problem is well known and the wrapper facade design pattern has been proposed as one solution [7, 8]. Wrapper facade asks to wrap OS specific call sequences in higher level method calls and to present them to an application as an abstract facade which is the same for all OSs. For each OS, the wrapping must be programmed once, and the middle-ware code presenting the facade must include some selection mechanism, called the adaptation, to choose the proper wrapper at hand. Two such selection mechanisms are proposed by Schmidt *et. al.* [8], and the purpose of the paper is to present a third one based on generic programming techniques, which we hold superior.

The rest of the paper is organized as follows. In Section 2 we review the wrapper facade design pattern which is used for defining abstract object-oriented interfaces to access system functionality. Section 3 discusses general portability issues of this approach. In Section 4 two standard techniques for mapping the abstract interface to target platforms are reviewed. Section 5 introduces our new technique for carrying out the mapping. A comparison of the presented techniques is the topic of Section 6.

2 The Wrapper Facade Design Pattern

All methods which we are discussing subsequently are based on the wrapper facade design pattern [7, 8]. In this section we give a brief overview of the main structure and participants of this design pattern. A more comprehensive

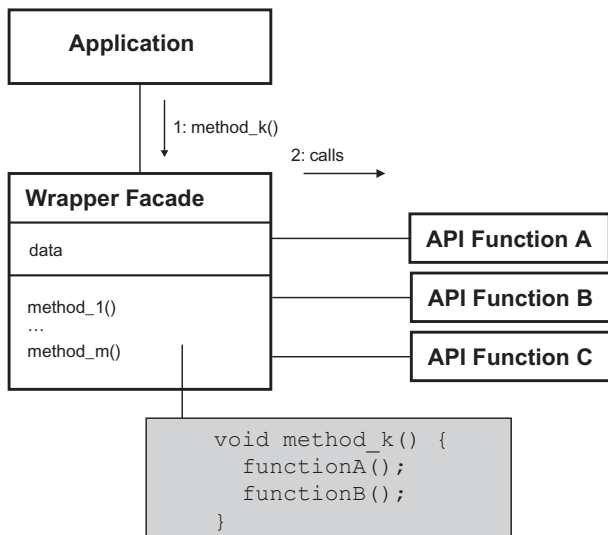


Figure 1. The Wrapper Facade Design Pattern

description can be found in [8]. The fundamental principle underlying the wrapper facade design pattern is to encapsulate the C based low-level system calls, library functions and data structures of operating system APIs within suitable class interfaces and hierarchies. This strategy enables application programmers to access system functionality through a more abstract object-oriented interface. Figure 1 depicts the following main participants of the wrapper facade design pattern.

- **OS API Functions**

A set of system calls, library functions and related data structures of a given target platform.

- **Wrapper Facade**

One or more classes that encapsulate elements of the system call interface. The classes integrate these elements into a more abstract interface.

The wrapper facade design pattern provides precise recipes on how to collocate system calls and related data structures according to their functionality into groups (e.g. multithreading, networking, ...) establishing cohesive abstractions and on how to design classes and class hierarchies that provide corresponding abstract interfaces.

However, the wrapper facade design pattern mainly concentrates on the process of designing appropriate abstract class interfaces and does not deal sufficiently with mapping these interfaces to target platforms with fundamentally different native APIs. In this paper we elaborate on portability aspects of the wrapper facade approach, i.e. the issue of the efficient implementation of abstract interfaces on a wide range of different target platforms. For the rest of the paper, we assume that according to the rules of the wrapper facade design pattern a class interface has already been defined introducing suitable abstractions for the given application area.

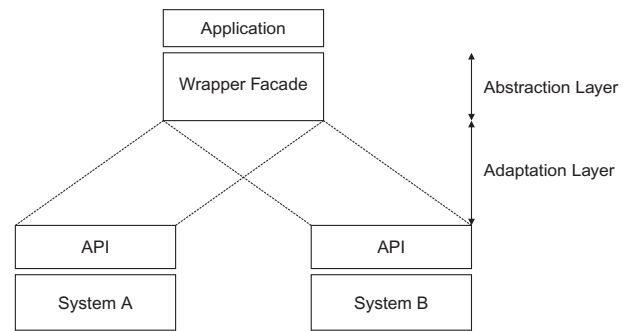


Figure 2. Wrapper Facade and Adaptation Layer

3 General Requirements for Achieving Portability of a Wrapper Facade

Conceptually, besides the *abstraction* software layer provided by the wrapper facade, an *adaptation* software layer has to be introduced, which is located between the wrapper facade interface and the native API. The role of the adaptation layer is to organize platform specific code that implements the abstract interface by employing a specific programming technique. Figure 2 illustrates the relationship between these two software layers.

Basically, abstract class interfaces encapsulate three different kinds of entities: methods, constants, and type definitions. Consequently, the technique for realizing the adaptation layer should provide support for efficiently mapping all these entities between the abstract interface and the corresponding native APIs.

In the next section we review two widely used techniques. Section 5 deals with our technique for implementing the adaptation layer which is based on generic programming. For the presentation of all techniques we use an exemplary class interface (`Abstract_Interface`). This wrapper facade encapsulates the method `Y syscall(X)` along with appropriate type definitions for the argument `X` and the result `Y` of `syscall` plus a constant named `xconst` of type `X`. Despite being very simple, this example interface comprises all relevant entities (methods, constants and types) occurring in abstract class interfaces.

4 Standard Techniques for Organizing the Adaptation Layer

In this section we briefly summarize two methods for realizing the adaptation layer that are well known and widely used. At their core, they are each based on one of the following language features:

- Conditional Compilation
- Virtual Methods

```

// type definitions
#ifdef SYS_A
    typedef sysa_x_t X;
    typedef sysa_y_t Y;
#endif

#ifdef SYS_B
    typedef sysb_x_t X;
    typedef sysb_y_t Y;
#endif

// constants
#ifdef SYS_A
    const X xconst = sysa_x_const;
#endif

#ifdef SYS_B
    const X xconst = sysb_x_const;
#endif

// methods
class Abstract_Interface {
public:
    static Y syscall(X arg) {

#ifdef SYS_A
        return ::syscall_A(arg);
#endif

#ifdef SYS_B
        return ::syscall_B(arg);
#endif
    }
};

```

Figure 3. Abstraction Layer Realized with Conditional Compilation

4.1 Conditional Compilation

Conditional compilation is the most commonly employed method for realizing portable programs. It uses preprocessor directives for conditional compilation (`#ifdef`) to select platform specific code according to the definition of constants that indicate specific properties of the considered target platform. Autoconfiguration tools (e.g. GNU autoconf [9]) can be used to automatically define appropriate preprocessor constants for a platform. Figure 3 shows the implementation of our exemplary abstract interface introduced in Section 3 using conditional compilation. Note that the conditional compilation technique establishes no clear distinction between abstraction and adaptation software layers at the source code level.

4.2 Virtual Functions

This technique represents an object-oriented approach for portable programming in C++. The abstraction layer com-

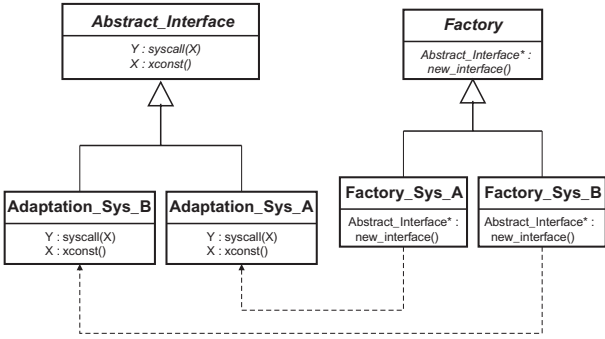


Figure 4. Abstract Factory Design Pattern

prises a set of abstract interfaces. In C++, abstract interfaces are classes that only include pure virtual functions. The adaptation layer is realized by providing for each target platform subclasses that implement virtual functions defined in the abstract interface.

Usually, in this scenario the instantiation of objects of appropriate platform specific classes is delegated to an analogously designed hierarchy of factory classes according to the abstract factory design pattern [10]. Figure 4 shows the structure of the abstract factory pattern in an UML class diagram.

Using factory classes to create platform specific objects has the advantage that locations in the code where distinctions between different platforms have to be made can be kept minimal. Ideally, at one location in the program, all platform specific factory objects are created. These factory objects can be used to transparently create platform specific objects that implement the abstract interface in all other parts of the application program. In order to further decouple the abstract class interface and its implementation the virtual function approach can be augmented by the Bridge design pattern [10]. This approach introduces two separate class hierarchies for abstraction and adaptation classes, but also fundamentally relies on the usage of virtual functions.

5 Implementation of the Adaptation Layer Using Generic Programming

The template feature of C++ [11] makes it possible to supply types as parameters for classes and methods. This programming technique is also called *generic programming* [12]. The main idea of employing generic programming to create portable C++ programs is to parameterize the abstraction layer with a suitable adaptation layer for the desired target platform. The classes of the abstraction layer are implemented in terms of abstract methods, constants and types which are encapsulated in platform-specific classes. Figure 5 shows the code for the exemplary abstract interface introduced in Section 3. In this example, preprocessor directives are used to create a platform specific interface. But in contrast to the conditional com-

```

template<class Adaptation>
class Abstract_Interface {
public:
    // type definitions
    typedef typename Adaptation::X X;
    typedef typename Adaptation::Y Y;

    // constants
    static const X xconst=Adaptation::xconst;

    // methods
    Y syscall(X arg) {
        return Adaptation::syscall(arg);
    }
};

#ifdef SYS_A
#include "Adaptation_Sys_A.h"
typedef
Abstract_Interface<Adaptation_Sys_A> OS;
#endif

#ifdef SYS_B
#include "Adaptation_Sys_B.h"
typedef
Abstract_Interface<Adaptation_Sys_B> OS;
#endif

```

Figure 5. Abstraction Layer Realized with Generic Programming

```

#include <sys/sysA.h>

class Adaptation_Sys_A {
public:
    // type definitions
    typedef sysa_x_t X;
    typedef sysa_y_t Y;

    // constants
    static const X xconst = sysa_x_const;

    // methods
    static Y syscall(X arg) {
        return ::syscall_A(arg);
    }
};

```

Figure 6. Realization of the Adaptation Layer with Generic Programming

pilation method, the preprocessor directives only occur at a single location in the source code.

Applying the generic programming technique, the class `Abstract_Interface` is realized as a template class, where the formal template argument is used to pass in an adaptation layer for a concrete operating system platform. Within the implementation of the abstract inter-

```

int main(int argc, char* argv[])
{
    OS* sys = new OS;

    // types
    OS::Y res;

    // constants
    const OS::X arg = OS::xconst;

    // functions
    res = sys->syscall(arg);

    std::cout << "res=" << res << std::endl;

    return 0;
}

```

Figure 7. Example Program.

face, the formal template argument is used to implement the methods, constants and type definitions (employing the `typedef` keyword) in terms of members of the formal template argument class.

The formal arguments are defined in the platform specific adaptation layer. Figure 6 shows the code of the adaptation layer for an exemplary native API. An example application code illustrating the usage of the abstract interface is depicted in Figure 7. Our technique is similar to the traits technique [13] which for example has extensively been used to implement the standard C++ library. But it introduces only one level of indirection.

6 Comparison

6.1 Criteria

We compare our approach based on generic programming presented in the last section with the two standard methods discussed in Section 4 according to the following criteria:

- **Efficiency**

Clearly, in the field of high performance computing, efficiency is the primary concern. The introduction of additional software layers should not impose a significant performance deterioration of applications. In the ideal case, the implementation technique for the adaptation layer permits the compiler to remove additional layers when generating object code.

- **Portability**

To be beneficial, the considered method should allow to easily integrate a large number of different target platforms. This task is greatly simplified when platform specific parts of code exhibit a high degree of locality, e.g. being separated by a syntactic unit. Moreover, changes to existing code that have to be made to

integrate a new target platform should only be minimal and should also exhibit a high degree of locality.

- **Enforcement of Correctness**

Modern operating systems provide a vast number of abstractions. Consequently, the corresponding application programming interfaces are extensive and complex. Necessarily, the same is true of the abstraction and adaptation layers. As an essential requirement for achieving correctness, it is indispensable that the completeness of the implementation of an abstract interface can already be verified at the compile time of the adaptation layer code. Otherwise, missing platform specific functionality could lead to compile errors when an application is built, or even worse, run time errors occur when the application is executed.

In order to carry out an exact comparison, we consider each of the three techniques in its pure form, where this is possible. In practice, it could also be feasible to use combinations of the techniques.

6.2 Efficiency

Besides inevitable conversions of data types between abstract and native formats, the major source of inefficiency occurring in all three techniques arises from additional software layers which lead to several indirections when accessing system functionality. Accessing system functionality through an abstract interface triggers a chain of nested function calls, where the innermost call is the actual system call.

The employment of the `inline` keyword of C++ can eliminate the overhead imposed by such function call nestings completely. Instead of generating a subroutine call, the compiler replaces a function call with the body of the corresponding function.

Not all of the presented techniques for realizing abstraction layers are compatible with the C++ `inline` feature. Conditional compilation and generic programming work well with this optimization. Thus the overhead of function call nesting can be eliminated completely for these two techniques using function inlining. However, inlining cannot be used in conjunction with virtual functions. A virtual function cannot be declared `inline`, because the actual method that is executed when a virtual function is called is determined at run-time (*late binding*). Moreover, the use of virtual functions to realize the adaptation layer itself has a negative impact on the application's performance. Each call to a virtual function requires first a lookup in the object's virtual function table (vtb) resulting in an additional type of indirection.

Result:

While the conditional compilation and the generic programming technique can be implemented efficiently, the virtual functions approach suffers from decreased performance caused by indirections.

6.3 Portability

The conditional compilation technique supports no locality of platform specific code, it is scattered throughout the adaptation layer. Thus, if a new target platform is to be integrated, modifications and extension to existing code have to be carried out at many different locations, not separated by any syntactical units. This property of the conditional compilation technique makes it extremely costly and error prone to add new target platforms. Its support for high portability is therefore limited.

The virtual functions method places platform specific implementations of methods declared in the abstract interface into different subclasses. However, this concept of redefinition does not extend to constants and type definitions. Consequently, for treating constants and types, another technique must be applied (e.g. conditional compilation) resulting in the drawbacks already described.

The generic programming technique provides for methods, as well as for constants and for type definitions a consistent way of completely separating platform specific parts of code into different syntactical units. All three entities can be grouped in one class which is used as actual template parameter when instantiating the abstraction layer.

Result:

The generic programming technique is the only method that completely supports locality of platform specific code and therefore allows a high degree of portability most effectively.

6.4 Enforcement of Correctness

As described above, with the conditional compilation technique, platform specific code is spread across the whole program code of the adaptation layer without being assigned to a distinct syntactical unit. So it is in general not possible to ensure the completeness of the implementation of the abstract interface for a given platform at compile time of the corresponding adaptation layer.

Employing the virtual functions approach it is possible to detect missing platform specific method implementations at compile time of the adaptation layer by testing if all interfaces can be instantiated. (If a subclass does not implement all virtual functions of an abstract interface it remains an abstract class and cannot be instantiated.) However, this property does not extend to the checking for definition of constants and types, since these entities must be realized using conditional compilation.

Using the generic programming technique, missing method, constant or type definitions can easily be detected at compile time of the adaptation layer by instantiating the classes of the abstraction layer with the corresponding classes of the adaptation layer for the desired target platform as actual template parameter. Missing definitions in the adaptation layer will result in a compiler error, regard-

less if it's a method, constant or type definition that is actually being absent.

Result:

The generic programming approach is the only technique that can ensure the complete implementation of all methods, constants as well as type definitions at the compile time of the adaptation layer.

7 Related Work

The commercial software library C++ ToolKits by Recursion Software, Inc., [14] provides abstract class interfaces for accessing system functionality for a wide range of operating system platform. For the mapping of the class interfaces on different target platforms the conditional compilation technique is used. The ADAPTIVE Communication Environment [15] toolkit provides similar class abstractions, implemented on top of an OS adaptation layer which uses conditional compilation. Another C++ cross-platform development toolkit based on conditional compilation is ZooLib [16].

8 Conclusion

In this paper we discussed techniques for creating highly portable programs in C++. All approaches are based on the wrapper facade design pattern, which is used to provide an abstraction software layer for homogenizing system functionality. For realizing a corresponding adaptation software layer, a novel technique, based on generic programming, is compared with two standard programming techniques: conditional compilation and the abstract factory design pattern.

The result of the comparison was that the generic programming technique for realizing the adaptation layer does not have drawbacks which are exhibited by the other two techniques. It represents the most suitable technique in terms of efficiency, portability and enforcement of completeness for realizing highly portable implementations of the wrapper facade design pattern in C++.

References

- [1] Ian Foster and Carl Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
- [2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.
- [3] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1995.
- [4] Wolfgang Blochinger, Wolfgang Küchlin, Christoph Ludwig, and Andreas Weber. An object-oriented

platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

- [5] Wolfgang Blochinger. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS 2001)*, page 90, San Francisco, CA, U.S.A., April 2001. IEEE Computer Society Press.
- [6] Wolfgang Blochinger, Reinhard Bündgen, and Andreas Heinemann. Dependable high performance computing on a Parallel Sysplex cluster. In Hamid R. Arabnia, editor, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, volume 3, pages 1627–1633, Las Vegas, NV, U.S.A., June 2000. CSREA Press.
- [7] Douglas C. Schmidt. Wrapper-facade – a structural pattern for encapsulating functions within classes. *C++ Report*, Feb 1999.
- [8] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2000.
- [9] GNU Autoconf. <http://www.gnu.org/software/autoconf>.
- [10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [11] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
- [12] Mehdi Jazayeri, Rüdiger Loos, and David Musser, editors. *Generic programming: International Seminar on Generic Programming, Dagstuhl Castle, Germany, 1998; selected papers*. Number 1766 in Lecture Notes in Computer Science. Springer Verlag, 2000.
- [13] Nathan C. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.
- [14] C++ toolkits. <http://www.recursionsw.com>.
- [15] Douglas C. Schmidt. ACE: An object-oriented framework for developing distributed applications. In *Proc. of the 6th USENIX C++ Technical Conference*, Cambridge, MA, April 1994. USENIX Association.
- [16] ZooLib. <http://zoolib.sourceforge.net>.