UDC 004.43=93C++
Original Scientific Paper

# Portable reflection for C++ with Mirror

**Matúš Chochlík** *Matus.Chochlik@fri.uniza.sk*
*University of Žilina*
*Faculty of Computer Science Žilina*

## Abstract

Reflection and reflective programming can be used in a broad range of tasks like implementation of serialization operations, remote procedure calls, scripting, automated user interface generation, implementation of several software design patterns, etc. C++ as one of the most prevalent programming languages however, for various reasons, lacks a standardized reflection facility. In this paper we present Mirror - a portable library adding reflection to C++ with a command-line utility automating its usage. This library supports functional style static compile-time reflection and metaprogramming and also provides two different object-oriented run-time polymorphic layers for dynamic reflection.

**Keywords:** reflection, reflective programming, metaprogramming, design-pattern implementation

## 1. Motivation

There are several distict computer programming tasks involving the execution of the same algorithm on a set of data structures or types defined by an application or on instances of these types, manipulating data members, global variables, calling free functions defined in namespaces or class member functions in a uniform manner, conversion of data between the programming language's intrinsic representation and various external formats for the purpose of implementing

- serialization or storing of persistent data in a custom binary format or in XML, JSON, XDR, etc. and deserialization or (re-)construction of class instances from external data representation formats, from the data stored in a RDBS, from the data entered by a user through a user interface,
- automatic generation of a relational schema from the application object model and object-relational mapping (ORM),
- support for scripting
- remote procedure calls (RPC) / remote method invocation (RMI),
- object inspection and manipulation via a GUI
- object access through web services (WS),
- visualization of application data and the relations in it,
- automatic or semi-automatic implementation of certain software design patterns,
- documentation or conceptual representation of a software system.

Historically, several different approaches were taken to implement the software features described above. The first - manual implementation, is also the most error-prone, since it involves writing new or changing the existing software source code in a tedious and inherently repetitive way. It basically requires processing programming language constructs like types (either atomic or elaborated), functions, constructors, class inheritance, class member variables, enumerated values, namespace members, or even high-level programming constructs like containers, etc. in a uniform way that could be easily algorithmized.

It may be acceptable even if not very advantageous for a design pattern [13] implementation to be made by a human programmer, but generating code related to RPC/RMI, scripting or ORM is a task much better suited for a computer. Other procedures like creation of documentation or knowledge representation of a software system, require some human work, but can be partially automated, which lead to the second, currently a very popular approach: preprocessing and parsing of the program source by a usually very specific external program (documentation generation tool, interface definition language compiler, interface generator for a REST web service, a support tool for ORM, a rapid application development environment with a form designer, etc.). The result is additional program source code, which is then compiled and integrated into the final application binary code.

While adequate in many situations, this approach has also several disadvantages: It requres external tools which may not fit well into the used build system or may not be portable across platforms, such tools are task-specific and most of them don't allow any or only a limited customization of the output.

Another possibility for automating the implementation of the features described above is to employ reflection, reflective programming, metaprogramming [2] and generic programming [3]. Automated implementation of the *factory* design pattern using reflection is shown for example in [15] (although the method described therein is somewhat limited because it requires that the constructed types have a default constructor and thus is not applicable to all types).

This technique is also used by several other reflection facilities, with similar limitations. Other examples of design pattern implementation with the help of reflection can be found in [30, 31, 33]. Reflection can also be employed to perform other tasks listed above as shown by projects like [1, 24, 25, 32] and others.

## 2. Introduction to reflection and related work

*Reflection* provides a computer program with the ability to observe and possibly alter its structure and/or its behavior. This includes altering the existing or building new data structures, doing changes to algorithms or changing the way the program code is interpreted [14]. Reflective programming is a particular kind of *metaprogramming*.

One of the advantages of reflection is that everything is implemented in a single programming language, and the human-written code can be closely tied with the reflection-based program code which is automatically generated by compiler metaprograms, based on the metadata provided by reflection.

Support for reflective programming is most common in high-level languages, often using a virtual machine, an interpreter or another such run-time environment, like JAVA, C#, CLOS or Smalltalk [6, 7, 11] and less common or limited in lower-level statically typed languages like Objective C, Lisp or Scheme [14].

Today, C++ as one of the most popular [21, 23] multi-paradigm programming languages, lacks a reflection facility. One of the attempts for standardized reflection in C++ was made in [29], but the work seems to be on hold recently [27]. Another recent attempt to add "rich pointers" and reflection to C++ [4] looks more promising, but does not address several important aspects like

strongly typed enumerations, template parameters, traversal of namespace members, etc. and is focused on run-time reflection.

There are several non-standard reflection systems, with varying degree of introspective and reflective capabilities, using various approaches.

The OpenC++ [22] is a programming toolkit that allows writing meta-level source-to-source transformation programs according to a meta-object-protocol (MOP) [7, 30] at the program pre-processing/parse-time. The meta-level program is compiled by the GCC C++ compiler linked with an OpenC++ add-on. Unfortunately only older compiler versions are supported which makes the approach non-portable and outdated. Other publications [34, 35] describe a similar source-to-source translator called *FOG*, adding support for aspect-oriented programming and basic compile-time reflection to C++. More common are run-time reflection systems like *Seal* C++ reflection system developed at CERN [27], or those described in [11, 12, 16, 17, 18, 26]. There are also several static reflection facilities with some run-time features, like the one described in [28].

## 3. Mirror reflection utilities

### 3.1 Introduction and history

Mirror implements a set of portable and non-intrusive compile-time and run-time introspection and reflection libraries and utilities for the C++ language. It provides reusable metadata reflecting C++ program constructs like namespaces, types, classes, base class inheritance, member variables, class templates and their template parameters, free functions, etc. and implements an extensive set of metaprogramming tools for the traversal and usage of these metadata in compile-time functional and run-time object-oriented algorithms. It also contains several high-level utilities, for example the factory generator utility, described in greater detail in [8]. It adheres to the three principles attributed to well-designed reflective facilities [6]: *encapsulation*, *stratification* and *ontological correspondence*.

The library and the related tools are implemented as proof-of-concept for a future proposal for standardised reflection for the C++ language and the source code is available for download under the Boost Software License [5] from SourceForge.net at the following URL: `http://sourceforge.net/projects/mirror-lib/files/`.

The current version [19], which is in active development, is using several new features from the current ISO standard for the C++ language (C++2011). This version removes several design problems discovered during the development and usage of the previous versions and in addition contains a run-time layer, built on top of the compile-time metaobjects, providing a dynamic, polymorphic object-oriented interface for their usage in run-time reflective algorithms. Recently there have also been added both an object-oriented compile-time layer and a type-erasure utility. This quaternary interface makes the library applicable in a wide range of scenarios.

### 3.2 Architecture

Mirror has a hierarchic multi-tier architecture and each layer provides services to the layers above as shown on Figure 1.

- *Registering and basic metadata*: Standard C++ provides only a limited set of metainformation to build upon, therefore the basic programming language constructs like namespaces, types, typedefs, classes, variables, functions, constructors, enumerated values, operators, etc. need to be registered before they can be reflected. Mirror tries to make the process of registering simple and convenient by providing a set of user-friendly registering macros and already has the intrinsic types and many of the other common types, classes, templates and namespaces pre-registered. This is the lowest layer and is used by the applications only to

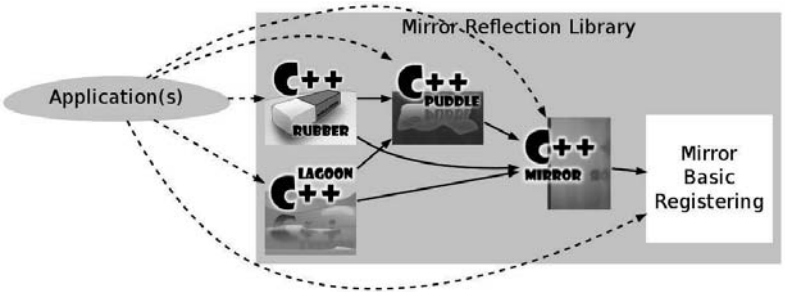


Figure 1: Architecture of the Mirror library

register their components. This registering should be replaced in future by static meta-data provided by the compiler similar to the now standardized `type_traits` as described for example in [9].

- *Functional compile-time layer* - **Mirror**: Mirror is a compile-time functional-style reflective programming library, which is based directly on the basic metadata and is suitable for generic programming, similar to the standard `type_traits` library. It allows to write compile-time metaprograms which enable the compiler to generate efficient program code.
- *Object-oriented compile-time layer* - **Puddle**: Puddle is a OOP-style compile-time interface with some run-time features built on top of Mirror. It copies the metaobject concept hierarchy of Mirror, but provides a more object-oriented interface. It is still inherently static, allowing for extensive optimization by the compiler.
- *Object-oriented type-erasure utility* - **Rubber**: Based on the object-oriented compile-time layer; its purpose is to remove the exact types of the instances of the metaobject concepts provided by the lower layers and merge them into a single quasi-polymorphic type for each compile-time metaobject concept. This enables the type-erased metaobjects to be stored in standard containers (like vectors, maps, sets, etc.) and used in non-template functions (for example C++ lambda functions). Unlike the Lagoon layer, it does not employ virtual functions to achieve polymorphism. This utility is suited for situations where a combination of compile-time and run-time reflection is required.
- *Object-oriented run-time layer* - **Lagoon**: Based on top of the Mirror and Puddle layers, it provides a run-time polymorphic interface, more suitable for run-time reflective programming, allowing the use of the provided metadata in a dynamic manner dependent on other data available only at run-time. One of its disadvantages is the performance penalty induced by virtual function calls and the inability of the compiler to inline such calls in many cases together with long compilation times. In the future this layer will allow to compile the metainformation into shared dynamic libraries separate from the applications and load them on-demand.

### 3.3 Goals and Features

The libraries and related utilities are developed with the following goals in mind:

- *Reusability*: The metadata provided by Mirror is reusable in many situations and for many different purposes.
- *Flexibility*: Mirror and the additional layers built on top of it allow to access the provided metadata both at compile-time and run-time in a functional and object-oriented manner depending on the application needs.

- *Encapsulation*: Mirror and the additional layers provide interfaces for easy access to program metadata.
- *Stratification*: It is non-intrusive and separates the meta-level from the base-level program features. Things that are not needed are generally not compiled-into the final application.
- *Ontological correspondence*: The meta-level facilities correspond to the ontology of the base-level C++ language constructs which they reflect.
- *Completeness*: Mirror tries to provide as much useful metadata as possible, including various specifiers, iteration of namespace members and much more.
- *Ease of use*: Although it allows to do very complicated reflective metaprogramming, simple things are kept simple.
- *Cooperation with other libraries*: Mirror and the additional layers can be used with the introspection facilities provided by the standard library and third-party libraries.

### 3.4 Metaprogramming utilities

#### *3.4.1 Compile-time*

The metaprogramming tools implemented by Mirror allow writing complex algorithms, using metadata describing the program, executed by the compiler at compilation time resulting only in compile-time constants, types or program code chunks in intermediate representation that are afterwards incorporated into the handwritten program code that is being compiled. The following program code snippet shows one possible usage: processing all members of the global scope filtering out only types which have the string 'long' in their name, resulting in a series of calls to a lambda function printing the names of the types to the standard output.

```
using namespace mirror;
std::cout << "Types having 'long' in their name:" << std::endl;
mp::for_each<
    mp::only_if<
        members<MIRRORED_GLOBAL_SCOPE()>,
        mp::and_<
            mp::is_a<
                mp::arg<1>,
                meta_type_tag
            >,
            cts::contains<
                static_name<mp::arg<1> >,
                cts::string<'l','o','n','g'>
            >
        >
    >
>(
    [](const rubber::meta_named_object& type)
    {
        std::cout << type.base_name() << std::endl;
    }
);
```

This example is rather synthetic, but there are also real-life scenarios where this feature could be used. If a consistent naming policy is defined for a large application (like the names of all interfaces end with `Intf` or start with `I`, i.e. `ComponentIntf` or `IComponent`, or the names of all classes mapped to database tables starting with `DB`, etc.) special set of classes, functions, etc. can be selected and processed by a meta-program.

#### *3.4.2 Run-time*

The *Lagoon* layer implements run-time counterparts for the compile-time utilities from Mirror. The code in the following example extracts all types from the global scope, sorts them by their instance size and prints their names:

```
using namespace lagoon;
typedef shared<meta_type> shared_mt;

for_each(
    sort(
        extract<meta_type>(
            reflected_global_scope()->members()
        ),
        [](const shared_mt& a, const shared_mt& b)
        {
            return a->size_of() < b->size_of();
        }
    ),
    [](const shared_mt& member)
    {
        std::cout << member->full_name() << std::endl;
    }
);
```

### 3.5 High-level utilities

In addition to the metaobjects reflecting various language constructs described above, Mirror also provides several high-level reflection-based utilities.

#### *3.5.1 Factory generator*

The factory generator utility allows to automate the implementation of the *factory* design pattern for types registered by the Mirror library. By factory we mean here a class, which can create instances of a *product* type, but does not require that the caller chooses the manner of the construction nor supplies the required parameters directly in the native data representation of C++.

A factory generated by Mirror examines the input which can be provided in an external data representation like XML, JSON, XDR, a simple scripting language, a dataset which results from a query to a RDBS, a user interface, etc., selects the constructor that matches to the available input data, converts the data from the external representation and calls the constructor. It may even construct some of the arguments recursively by the means of other, nested factories.

The advantage of this reflection-based approach is, that it separates (from the programmer's point of view) and then automatically combines the parts specific to the construction of a particular type from the general logic of the constructor selection, input data validation and conversion. This

way any reflectable type can be constructed from any input data format for which the conversion logic is implemented.

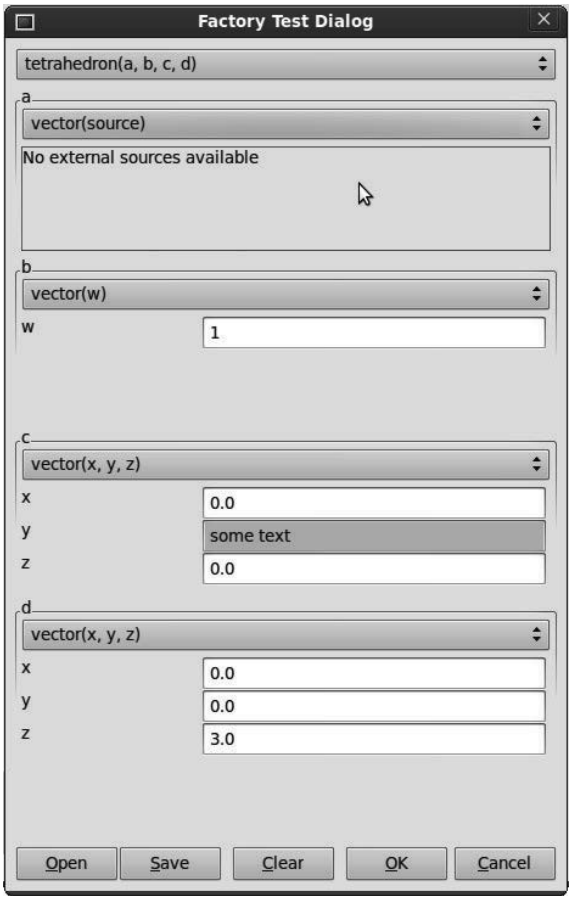


Figure 2: Example of a graphic user interface dialog constructed by an automatically generated factory.

The process of generation of the factories from the metadata provided by Mirror and a set of user-specified templates is described in greater detail in [8],[19]. Example of a GUI created by factory generated by the Mirror's factory generator for a simple `tetrahedron` class with the following definition, is shown on Figure 2.

```
struct vector
 {
     double x,y,z;

     vector(double _x, double _y, double _z);
     vector(double _w);
     vector(void);

     // ... + other declarations
 };

 struct triangle
 {
```

```
    vector a, b, c;

    triangle(vector _a, vector _b, vector _c);
    triangle(void);

    double area(void) const;
    // ... + other declarations
};

struct tetrahedron
{
    triangle base;
    vector apex;

    tetrahedron(const triangle& _base, const vector& _apex);
    tetrahedron(
        const vector& a,
        const vector& b,
        const vector& c,
        const vector& d
    );

    double volume(void) const;
    // ... + other declarations
};
```

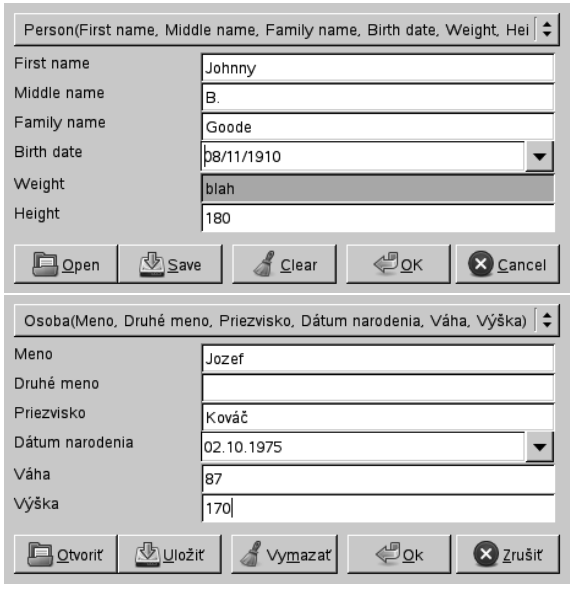


Figure 3: Examples of a GUI dialogs with enabled localization constructed by a factory generated by Mirror.

The automatically generated factory class creates a reusable GUI dialog window which allows the user to select any combination of constructors of the classes declared in the code shown above,

and provides means to input the values necessary to construct an instance of the `tetrahedron` class.

The dialog contains widgets for the input of the constructor parameters with atomic types (in this case `double`) with validators rejecting input of any text not convertible to a floating-point value and checking if all required data was provided. Upon clicking the OK button, the factory creates a new instance of the `tetrahedron` class and returns control to the application. This process can be repeated multiple times without the need to recreate the input dialog. When the factory no longer required it takes care of properly freeing the resources associated with the dialog window.

Figure 3 shows another two dialogs created by an automatically generated factory, for a `person` class. In this example the support for localization is enabled (with the `en_US` and `sk_SK` locales) which results in more user-friendly dialogs by translating the basic C++ identifier names on the label widgets to a more human-readable form and adapting the input widgets to regional format.

The following code shows an example where a generated factory constructs a complex type (a `std::list` of `tetrahedrons`) from a string in the JSON (JavaScript Object Notation) format. In this case the polymorphic factory generator from the *Lagoon* layer uses the `mijson` parser that comes with the Mirror library to parse the input text. The *polymorphic* factory builder allows to completelly decouple the meta-data from the factory builder and allows to specify both the reflected type and the input-data-handling part of the factory at run-time.

```
int main(void)
{
    try
    {
        using namespace lagoon;

        // a polymorphic factory builder using
        // the mijson parser
        mijson_factory_builder builder;

        // input handler for the factory
        mijson_factory_input in;

        // input data wrapper
        auto data = in.data();

        // the constructed type
        typedef std::list<test::tetrahedron> tetrahedron_list;

        // meta class reflecting the constructed type
        auto meta_tl = reflected_class<tetrahedron_list>();

        // the factory generated by the builder
        // and the meta class using the input
        // data wrapper
        auto tl_factory = meta_tl->make_factory(
            builder,
            raw_ptr(&data)
        );
```

```
// the input string in JSON containing data
// for the construction of several tetrahedrons
const char json[] =
    "["
    "   {"
    "       'base' : {"
    "           'a' : {'x': 1, 'y': 0, 'z': 0},"
    "           'b' : {'x': 0, 'y': 1, 'z': 0},"
    "           'c' : {'x': 0, 'y': 0, 'z': 0} "
    "       },"
    "       'apex' : {'x': 0, 'y': 0, 'z': 6} "
    "   },"
    "   {"
    "       'base' : {"
    "           'a' : {'x': 1, 'y': 0, 'z': 0},"
    "           'b' : {'x': 0, 'y': 2, 'z': 0},"
    "           'c' : {'w': 0} "
    "       },"
    "       'apex' : {'x': 0, 'y': 0, 'z': 6} "
    "   },"
    "   {"
    "       'base' : {"
    "           'a' : {'x': 2, 'y': 0, 'z': 0},"
    "           'b' : {'x': 0, 'y': 2, 'z': 0},"
    "           'c' : null "
    "       },"
    "       'apex' : {'x': 0, 'y': 0, 'z': 6} "
    "   },"
    "   {"
    "       'a' : {'x': 2, 'y': 0, 'z': 0},"
    "       'b' : {'x': 0, 'y': 2, 'z': 0},"
    "       'c' : { },"
    "       'd' : {'x': 0, 'y': 0, 'z': 12} "
    "   }"
    "]";

// a parser parsing the JSON string
mirror::mijson_fragment<const char*> fragment(
    json,
    json + sizeof(json) - 1
);

// use the parsed JSON fragment as input
in.set(fragment);

// use the factory to create a list of tetrahedrons
raw_ptr ptl = tl_factory->new_();

// cast the raw pointer to a typed pointer
```

```
        tetrahedron_list& tl = *raw_cast<tetrahedron_list*>(ptl);

        // traverse the constructed list of tetrahedrons
        for(auto i = tl.begin(), e = tl.end(); i != e; ++i)
        {
            // print some information
            std::cout
                << "the volume is "
                << i->volume() << " " << std::flush;
            std::cout
                << "the area of the base is "
                << i->base.area() << std::endl;
        }
        // use the meta class to properly destroy
        // the constructed instance of the list
        // of tetrahedrons
        meta_tl->delete_(ptl);
    }
    catch(std::exception const& error)
    {
        std::cerr << "Error: " << error.what() << std::endl;
    }
    //
    return 0;
}
```

This application produces the following output:

```
the volume is 1 the area of the base is 0.5
the volume is 2 the area of the base is 1
the volume is 4 the area of the base is 2
the volume is 8 the area of the base is 2
```

Note that each tetrahedron in the input string is constructed with a different set of `tetrahedron` and `vector` constructors.

This application could be extended for example to connect to a REST (REpresentational State Transfer) web service hosted on a network, providing data in JSON format and then convert and process the incomming data in C++. Mirror also provides factory builders parsing XML and a C++-like script language and builders loading data from RDBS datasets using the `libpq` and `soci` libraries (see [19] for more examples).

#### *3.5.2 Invoker generator*

The invoker generator allows to create invoker classes which are able to call arbitrary reflectable functions, in the same manner as the generated factories call constructors. This feature is still experimental.

### 3.6 External tools

The process of manual registering has several advantages, but can also be tedious and error-prone. The registering macros use auto-detection where possible and many things do not have to

be specified explicitly, but some changes in the base-level classes, like adding or removing of a member variable, constructor or a whole class, etc. still require updates of the registering code.

If no special customization in the reflection of the base-level constructs (like hiding certain members or constructors of a class or specifying of getter/setter functions for a member variable) is required, then support for automatic reflection is desirable.

The *MAuReEn* (Mirror Auto-Reflection Engine) project also developed by the author, generates the registering code automatically while still allowing significant customization. This tool parses the header files containing declarations of the base-level constructs to be registered (namespaces, types, classes, variables, etc.) and outputs the required registering code into header files as specifed by the user. It can be easily integrated into existing build systems like GNU Make, CMake, Boost.Build and others. It has a modular architecture which allows to implement various methods of input file parsing and output file generation. The automatically generated registering code can be easily combined with hand-written registering code. The sources for this project are available at `http://gitorious.org/maureen`. In future this tool could be completely replaced by standard reflection as described in [9].

## 4. Conclusion and future work

As shown above, reflection can be a valuable programming tool. Even if excellent for compile-time metaprogramming, the standardized C++ language misses a reflection facility. The goal of the Mirror reflection utilities is to address this shortcoming and to provide a proof-of-concept implementation of portable reflection for C++. Mirror is still in development and there are several features that are planned for future releases, including but not limited to the following:

- Refactoring of the existing facilities for registering and reflection of free and class member functions with proper support for function overloads.
- An abstract factory generator that would combine the concrete factories generated by the existing factory generator utility into a single abstract factory.
- An object manipulator generator that would allow generating manipulators working on existing instances of various types. Such manipulators could perform tasks like GUI object inspection, serialization, etc. in a similar manner as the generated factories are used for object instantiation.
- An additional semantic-layer that would allow to tie conceptual data to the reflected program components, allowing the visualization of application's structure and data, generation of more user-friendly GUI/Web interfaces, access for agent oriented systems and automated reasoning.

## References

[1] A C++ reflection-based data dictionary, http://sourceforge.net/projects/crd/

[2] Abrahams, D; Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004

[3] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[4] Berris, M. D.; Austern, M.; Crowl, L.: Rich Pointers. Proposal for ISO/IEC JTC1 SC22 WG21, N3340=120030, 2012, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3340.pdf

[5] Boost Software License, Version 1.0, http://www.boost.org/LICENSE_1_0.txt

[6] Bracha, G; Ungar, D. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 85 104, 2003.

[7] Chiba, S. A Metaobject Protocol for C++. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1995.

[8] Chochlík, M. Generating Object Factory Classes with the Mirror Reflection Library. *Journal of Information, Control and Management System*, Vol. 8, No. 2, ISSN 1336-1716, 2010.

[9] Chochlík, M. Static reflection. Proposal for ISO/IEC JTC1 SC22 WG21, C++ Programming language, Library working group, 2012, http://kifri.fri.uniza.sk/ chochlik/jtc1_sc22_wg21/std_cpp_refl.pdf

[10] Chochlík, M. Support for object-oriented parallel programs for grids and clusters. *Proceedings of 2nd International Workshop on Grid Computing for Complex Problems*, Bratislava, Slovakia, pages 88-96, 2006.

[11] Chuang, T-R; Kuo, Y.S; WANG, C-M. Non-intrusive object introspection in C++. *Software-Practice and Experience*, issue 32, pages 191-207, 2002.

[12] Devadithya, T; Chiu, K; Lu, W. C++ Reflection for High Performance Problem Solving Environments. *In Proceedings of the 2007 spring simulation multiconference*, Volume 2, Norfolk, Virginia, USA, pages: 435-440, 2007.

[13] Gamma, E; Helm, R; Johnson, R; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Proffesional, ISBN 0-201-63361-2, 1995.

[14] Kirby, G.N.C. *Reflection and Hyper-Programming in Persistent Programming Systems*, Ph.D. Thesis, University of St Andrews, 1992.

[15] Kovacs, R. *Creating Dynamic Factories in .NET Using Reflection*, http://msdn.microsoft.com/en-us/magazine/cc164170.aspx

[16] Madany, P. W; Islam, N; Kougiouris, P; Campbell, R. H. Reification and Reflection in C++: An Operating Systems Perspective. 1992.

[17] Madina, D; Standish, R. K. A system for reflection in C++. *Proceedings AUUG 2004: Always on and Everywhere*, 2004.

[18] Metaclasses and Reflection in C++, http://www.vollmann.ch/pubs/meta/meta/meta.html

[19] Mirror C++ reflection library documentation (C++11 version), http://kifri.fri.uniza.sk/ chochlik/mirror-lib/html/

[20] MPI: A Message-Passing Interface Standard. MPI Forum, 2003, http://www.mpi-forum.org/docs/mpi1-report.pdf

[21] Ohloh.net: compare languages tool, http://www.ohloh.net/languages/compare

[22] OpenC++, http://opencxx.sourceforge.net

[23] Programming Language Popularity, http://langpop.com/

[24] Property Set Library (PSL), http://sourceforge.net/projects/psl/

[25] QxOrm library, http://sourceforge.net/projects/qxorm/

[26] Reflection for C++, http://www.garret.ru/cppreflection/docs/reflect.html

[27] Roiser, S; Mato, P. The Seal C++ Reflection system. CERN, Geneva, Switzerland.

[28] Static reflection in C++ using minimal repetition, http://www.enchantedage.com/cpp-reflection

[29] Stroustrup, B. XTI An Extended Type Information Library, http://lcgapp.cern.ch/project/architecture/XTI_accu.pdf

[30] Tanter, E; Noyé, J; Caromel, D; Cointe, P. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. *Proceedings OOPSLA'03*, Anaheim, California, USA, 2003.

[31] Tatsubori, M; Chiba, S. Programming Support of Design Patterns with Compile-time Reflection. *Proceedings OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, 1998.

[32] Template Reflection Library, http://sourceforge.net/projects/trl/

[33] The Visitor Pattern, http://www.oodesign.com/visitor-pattern.html

[34] Willink, E. D; Muchnick, V. B. Weaving a Way Past the C++ One Definition Rule. *Proceedings of European Conference on Object Oriented Programming*. Lisbon, June 14, 1999.

[35] Willink, E. D. Preprocessing C++: Meta-Class Aspects. *Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems*, TOOLS EE 99, Blagoevgrad, Bulgaria, June 1999.