STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 1, 2011

ADVANCED FUNCTOR FRAMEWORK FOR C++ STANDARD TEMPLATE LIBRARY

NORBERT PATAKI

Abstract. The *C++ Standard Template Library (STL)* is the most popular library based on the *generic programming paradigm*. STL is widely-used, because it consists of many useful generic data structures and generic algorithms that are fairly irrespective of the used container. Iterators bridge the gap between containers and algorithms. As a result of this layout the complexity of the library is reduced and we can extend the library with new containers and algorithms simultaneously.

Function objects (also known as *functors*) make the library much more flexible without significant runtime overhead. They parametrize user-defined algorithms in the library, for example, they determine the comparison in the ordered containers or define a predicate to find. Requirements of relations are specified, for instance, associative containers need strict weak ordering. However, these properties are tested neither at compilation-time nor at run-time. If we use a relation that is not a strict weak ordering, the containers become inconsistent. Only *adaptable functors* are able to work together with function adaptors. Unfortunately, the adapted functors type requirements come from special typedefs. If these typedefs are errounous ones, the adapted functor does not work perfectly.

In this paper we present our framework that aims at developing safe adaptable functors. One of the characteristics tested at runtime, some of them handled at compilation-time. This framework is based on the object-oriented and generative features of C++. Our aim is to develop a safe, efficient, multicore version of C++ STL in the future.

1. Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [3]. In this way containers are defined as class templates

Received by the editors: March 24, 2011.

2010 *Mathematics Subject Classification.* 68N15, 68N19.

1998 *CR Categories and Descriptors.* D.2 [**Software Engineering**]: D.2.5 Testing and Debugging – *STL functors*; D.3 [**Programming Languages**]: D.3.2 Language Classification – *C++*.

Key words and phrases. C++, STL, functors.

and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way [15], so one can use them with different containers [24]. C++ STL is widely-used because it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.), large number of algorithms (like sort, find, count, etc.) among other utilities.

The STL was designed to be extensible. We can add new containers that can work together with the existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can work together with the existing containers. Iterators bridge the gap between containers and algorithms [5]. The expression problem [26] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [1].

However, the usage of C++ STL does not mean bugless or error-free code [8]. Contrarily, incorrect application of the library may introduce new types of problems [23].

One of the problems is that the error diagnostics are usually complex, and very hard to figure out the cause of a program error [27, 28]. Violating requirement of special preconditions (e.g. sorted ranges) is not tested, but results in runtime bugs [11, 19]. A different kind of stickler is that if we have an iterator object that pointed to an element in a container, but the element is erased or the container's memory allocation has been changed, then the iterator becomes *invalid* [9]. Another common mistake is related to removing algorithms. The algorithms are container-independent, hence they do not know how to erase elements from a container, just relocate them to a specific part of the container, and we need to invoke a specific erase member function to remove the elements phisically. Since, for example the `remove` algorithm does not actually remove any element from a container [14].

C++ STL is very efficient in a sequential realm, but it is not aware of multicore environment [4]. For example, the Cilk++ language aims at multicore programming. This language extends C++ with new keywords and one can write programs for multicore architectures easily. But the language does not consist of an efficient multicore library, but the C++ STL only which is an efficiency bottleneck in multicore environment. We develop a new STL implementation for Cilk++ to cope with the challenges of multicore architectures. This new implementation can be safer solution, too. Hence, our safety extensions will be included in the new implementation. However, the advised techniques presented in this paper concern to the original C++ STL, too.

Most of the properties are checked at compilation time. For example, the code does not compile if one uses sort algorithm with the standard list container, inasmuch as list's iterators do not offer random accessibility [12].

Other properties are checked at runtime. For example, the standard vector container offers an `at` method which tests if the index is valid and it raises an exception otherwise [18].

Unfortunately, there is still a large number of properties that are tested neither at compilation-time nor at run-time. Observance of these properties is in the charge of the programmers.

Functor objects make STL more flexible as they enable the execution of user-defined code parts inside the library. Basically, functors are usually simple classes with an `operator()`. Inside the library `operator()`s are called to execute user-defined code snippets. This can called a function via pointer to functions or an actual `operator()` in a class. Functors are widely used in the STL because they can be inlined by the compilers and they cause no runtime overhead in contrast to function pointers. Moreover, in case of functors extra parameters can be passed to the code snippets via constructor calls.

Functors can be used in various roles: they can define predicates when searching or counting elements, they can define comparison for sorting elements and properly searching, they can define operations to be executed on elements.

Associative containers (e.g. `multiset`) use functors exclusively to keep their elements sorted. Algorithms for sorting (e.g. `stable_sort`) and searching in ordered ranges (e.g. `lower_bound`) typically used with functors because of efficiency. These containers and algorithms need *strict weak ordering* [2].

A relation is strict weak ordering, if it is irreflexive, antisymmetric and transitive. A relation is irreflexive, if no element is related to itself. A relation is antisymmetric, if two elements are in relation and these two elements are in relation in reverse order means that two elements are the same. A relation is transitive if whenever an element a is related to an element b, and b is in turn related to an element c, then a is also related to c. For example, internal types' `operator<` is strict weak ordering relation in contrast to their `operator>=`, that are not strict weak ordering.

The rest of this paper is organized as follows. In section 2 we show the necessity of a framework to develop safe functors. We present our approach for testing functors at runtime in section 3. We detail how to develop safe adaptable functors with our framework in section 4. Finally, we conclude our results in section 5.

2. Motivation

In this section we present motivating examples. We detail the background of the problems because they are not evident.

All standard associative containers (`std::set`, `std::multiset`, `std::map`, `std::multimap`) keep elements ordered. These containers are template classes – among other template parameters they have a type parameter which stands for the type of comparison functor. That is, in this case it is not possible to pass function pointers. We must use functor types instead. From this type the containers create object to evaluate the comparison between elements. This parameter has default value which is `std::less<T>`, where T is the key type of the container. The functor must be strict weak ordering, but this property is not checked, so it is not difficult to write an erroneous one [16], for example:

```
struct Compare :
         std::binary_function<int, int, bool>
{
  bool operator()( int i, int j ) const
  {
    return !(i < j);
  }
};

struct StringLengthLess :
         std::binary_function<std::string, std::string, bool>
{
   bool operator()( const std::string& a,
                    const std::string& b ) const
   {
      return a.length() <= b.length();
   }
};
```

These functor types can be compiled, and they do not raise exception or assertation at runtime, too. On the other hand, neighter is strict weak ordering, especially as they do not meet the requirement of irreflexivity. The usage of these functors makes associative containers inconsistent:

```
std::set<int, Compare> sc;
sc.insert( 3 );
sc.insert( 3 );
// sc.size() == 2
// sc.count( 3 ) == 0

std::multiset<int, Compare> mc;
mc.insert( 7 );
// mc.count( 7 ) == 0
```

Many standard algorithms can be used for searching in ordered ranges, for instance `binary_search`, `lower_bound`, etc. All of these algorithms are overloaded and they can take a comparison functor as a parameter. These algorithms require strict weak ordering as well.

```
std::vector<int> v;
v.push_back( 4 );
v.push_back( 5 );
// ...
std::sort( v.begin(), v.end(), Compare() );

std::vector<int>:: iterator i =
  std::lower_bound( v.begin(), v.end(), 4, Compare() );

if ( i != v.end() )
{
  std::cout << "Not found";
}
```

When we use the erroneous functors, these algorithms cannot find elements in containers, hence they seem to be defective.

However, these examples are the most simplest ones. In case of more complex classes it is much easier to make a mistake.

As a general rule, two comparisons are used in the STL. One of them is called *equality*, and this is based on `operator==`. Algorithms, like `find` use equality when searching. Another one is called *equivalence* and this one is based on the relative ordering of object values in a sorted range. Two objects `x` and `y` have equivalent values with respect to the sort order if neither precedes the other in the sort order, so the following expression is evaluated: `!keycomp()(x,y) && !keycomp()(x,y)`, where `keycomp` is the type of the functor.

Let us consider what happens if the strict weak ordering requirement is violated, for example, the functor is based on `operator<=`. When elements are inserted into a `std::set`, the container has to examine if the element is already in the container. The container's `insert` method uses *equivalence*.

```
s.insert( 3 );
s.insert( 3 );
```

The second call of `insert` evaluates the following expression to check if value 3 is in the set:

```
!(3<=3) && !(3<=3)
```

The result of this expression is false, that can be interpreted as "3 is not equal to 3". This fact accounts for the previous strange behaviour of associative

containers. The container finds that 3 is not in the set, so it inserts once more. The set is not set anymore. The set's `count` member function cannot find any element that is equal to its parameter. The associative containers become inconsistent this way [pirkelbauer:runtime. This problem itself is not specific to C++, similar mistakes can be made in Java [21].

Adaptable functors are special functors which can be used with functor adaptors, such as `not1` or `bind2nd` [14]. Binder allow us to convert a binary function to a unary function, by binding one of the arguments to a given value (given at runtime). `not1` negates the unary predicate, `not2` negates the binary predicate, respectively.

Adaptable functors need some extra typedefs, and the adapted functors are generated based on these typedefs. The standard base type templates `unary_function` and `binary_function` are responsible to guarantee these necessary typedefs. Thus, the author of a functor class is responsible to make sure if the adapted works properly. It would be more elegant if the these typedefs come from the functor's `operator()`. If the template arguments of `unary_function` or `binary_function` disagree to the functor's `operator()` it may also results in compilation-time or run-time errors.

Let us consider the following predicate:

```
struct AnotherBadPredicate: std::unary_function<int, bool>
{
  double x;

  AnotherBadPredicate( const double& d ): x( d ) {}

  bool operator()( const double& a ) const
  {
    return a < x;
  }
};
```

This functor works perfectly, unless it is used without adaptors. However, if we have a vector, and we try to find the first element that is *not* less than a given value, we can negate the previous predicate:

```
std::vector<double> v;
v.push_back( 2.5 );
v.push_back( 8.3 );

std::vector<double>::iterator i =
  std::find_if( v.begin(),
                v.end(),
```

```
                std::not1( AnotherBadPredicate( 2.3 ) );

if ( i != v.end() )
{
  std::cout << *i << std::endl;
}
```

This code snippets highlights that 8.3 is the first element in the vector which is not less than 2.3. This result is faulty because 2.5 is the first element that is not less than 2.5. However, the problem itself the template argument of base template class `std::unary_function<int, bool>` inasmuch as the generated negated predicate take the parameter as integer value. In the negated functor 2.5 as integer is 2 and it calls the original functor with this value. The original functor takes the parameter 2.0 as double value and `2.0 < 2.5` is true, so it returns true. The generated functor negates this result, thus `std::not1( AnotherBadPredicate( 2.3 )` return false when it takes 2.5 value. However, this root cause of this problem is the duplication. The author has to repeat the type of `operator()`'s argument types. If it disagrees it could make the negated functor erronous. The compiler should deduce the parameters of `operator()` to avoid this kind of problems. However, if the two different types cannot be converted it results in compilation errors. One of the aims of our framework is detect this problem at compilation time.

3. Safe functors

In this section we present our framework approach to test functors at runtime if they are strict weak ordering.

Our approach is based on object-orientation and inheritance because functors typically are inherited from `std::binary_function` or `std::unary_function` class templates, therefore this approach is plausible. Functor adaptors, such as `std::not2` to negate predicates, take advantage of some typedefs that comes from the base type. For instance, standard functor types are written this way. So we can take advantage of the automatic call of the base class's default constructor.

The type of functor is passed to the `strict_weak_ordering` class template as well as the type of parameter of its `operator()`. We force the base object to be the instance of subtype's class, which is the type of functor actually. We take advantage of `static_cast` which is able to do casts between pointer types. This is not a problem in this case because we designed this class to be superclass. We can call the functor's `operator()` this way. We assume, that the type `T` has default constructor. It is not serious restriction because most classes do have default constructor. Note, we cannot take advantage

of virtual functions or `dynamic_cast` because in constructors only the static type is available. Unfortunately, we cannot use `static_cast` without pointers because it would create a new object which has to be evaluated with our approach. This means an infinite recursion. Our approach is based on *curiously recurring template pattern (CRTP)* [1]. This pattern replaces the dynamic polymorphism with compile-time mechanism [7].

This way we create a test case that is evaluated when the functor object is constructed. If the testcase fails it throws an exception, otherwise, it does not mean necessarily that the functor itself is perfect. However, most of erronous functors fail on this test because typically the irreflexevity is violated, and our approach focuses on this characteristic. Moreover, specializations can be created for more complex testcases.

Only the `operator()` is called after a `static_cast` operator. This is quite reasonable and negligible overhead for safety functors [20].

So, First, we create a new exception type:

```
struct bad_functor_exception
{
  // ...
};
```

After we develop the essence of our approach:

```
template <class T, class functor_to_check>
struct strict_weak_ordering
{
  strict_weak_ordering()
  {
    if ( static_cast
           <functor_to_check*>( this )->
             operator()( T(), T() )
       )
    {
      throw bad_functor_exception();
    }
  }
};
```

The `strict_weak_ordering` template class is easy to use when one writes a new functor type. The new functor type must be inherited from the instantiated `strict_weak_ordering` class. This way this approach is non-intrusive.

```
struct Compare :
         std::binary_function<int, int, bool>,
         strict_weak_ordering<int, Compare>
```

```
{
  bool operator()( int i, int j ) const
  {
    // ...
  }
};
```

As mentioned before, specializations can be created for specific types. In the specializations more complex test cases can be evaluated. But if we increase the number of test cases the runtime overhead increases, too. Let us consider the following examples:

```
template <class functor_to_check>
struct strict_weak_ordering<int, functor_to_check>
{
  strict_weak_ordering()
  {
    functor_to_check* p = static_cast<functor_to_check*>( this );
    if ( p->operator()( 3,  3  ) ||
         p->operator()( 22, 22 ) )
    {
      throw bad_functor_exception();
    }
  }
};

template <class functor_to_check>
struct strict_weak_ordering<std::string, functor_to_check>
{
  strict_weak_ordering()
  {
    const std::string test = "Hello World";
    if ( static_cast
           <functor_to_check*>( this ) ->
             operator()( test, test ) )
    {
        throw bad_functor_exception();
    }
  }
};
```

External testing frameworks are also able to check these properties [17]. But external testing frameworks typically need external tools [6]. Our approach does not require any external tool, programmers have to design functors by inheritance which is straightforward as mentioned before. With our approach the average user of the library can write functors safely without any testing framework and difficult mathematical background. Our approach works perfectly automatically, too.

4. Adaptable functors

In this section we present how our framework can be used to write perfect adaptable functors.

Compilers cannot emit warnings based on the erroneous usage of the library. STLlint is the flagship example for external software that is able to emit warnings when the STL is used in an incorrect way [10]. We do not want to modify the compilers, so we have to enforce the compiler to indicate if an adaptable functor type is defective. However, `static_assert` as a new keyword is introduced in C++0x to emit compilation errors based on conditions, but no similar construct is designed for warnings.

```
template <class T>
inline void warning( T t )
{
}

struct IMPROPER_FUNCTOR_BASE
{
};

// ...

warning( IMPROPER_FUNCTOR_BASE() );
```

When the `warning` function is called, a dummy object is passed. This dummy object is not used inside the function template, hence this is an unused parameter. Compilers emit warning to indicate unused parameters. Compilation of `warning` function template results in warning messages, when it is referred and instantiated. No warning message is shown if it is not referred. In the warning message the template argument is referred.

Different compilers emit this warning in different ways. For instance, Visual Studio emits the following message:

```
warning C4100: 't' : unreferenced formal parameter
...
```

```
see reference to function template instantiation 'void
warning<IMPROPER_FUNCTOR_BASE>(T)'
being compiled

        with
        [
            T=IMPROPER_FUNCTOR_BASE
        ]
```

And g++ emits the following message:

```
In instantiation of 'void warning(T)
      [with T = IMPROPER_FUNCTOR_BASE]':
... instantiated from here
... warning: unused parameter 't'
```

Unfortunately, implementation details of warnings may differ, thus no universal solution to generate custom warnings.

This approach of warning generation has no runtime overhead inasmuch as the compiler optimizes the empty function body. On the other hand – as the previous examples show – the message refers to the warning of unused parameter, incidentally the identifier of the template argument type is appeared in the message.

C++ metaprogramming facilities are able to detect if the parameter types of the `operator()` suit to the template arguments of the base class [22]. They are not necessarily the same because references and constant references can be used as functor arguments, but in this case no references or constant references given as template arguments [14]. We can generate warnings with the previous approach [25].

First, we present the framework for unary functors that checks if the base type is proper for the `operator()`:

```
template<bool b, class Fun>
struct __WARNING
{
  __WARNING()
  {
    warning( IMPROPER_FUNCTOR_BASE() );
  }
};

template <class Fun>
struct __WARNING<true, Fun>
{
```

```
};

template <class Fun>
class __check_unary_adaptability
{
  typedef BOOST_TYPEOF(&Fun::operator()) f_type;

  typedef typename
    boost::mpl::at_c<
      boost::function_types::parameter_types<f_type>, 1>::type
    arg_type;

  __WARNING< boost::is_same<
      typename boost::remove_const<
        typename boost::remove_reference<arg_type>::type>::type,
      typename Fun::argument_type>::value, Fun > w;
};

#define CHECK_UNARY_FUNCTOR(F) __check_unary_adaptability<F>();
```

The utility class template `__WARNING` takes a compile-time boolean parameter, and if value of this parameter is true `__WARNING` does *not* generate warning. Otherwise it instantiates the warning generator function template, thus programmer gets a compilation warning. It is also takes the type of the functor to be presented in the generated error warning. The core class template is `__check_unary_adaptability` which extracts the type of parameter of the functor's `operator()` and named as `arg_type` [13]. It also retrieves the functor's inner typedef called `argument_type` which set by `unary_function`. Set type is template argument of `unary_function`. After that, it passes the condition of the two types are proper to the `__WARNING`. The proper parameter type means, that we should remove the `const` and `&` modifiers from the declaration of the parameter of `operator()`. Unfortunately, we have to start the verification manually, therefore a comfortable macro is present.

The following code snippets aims at the verification of binary functors:

```
template <class Fun>
class __check_binary_adaptability
{
  typedef BOOST_TYPEOF(&Fun::operator()) f_type;

  typedef typename
    boost::mpl::at_c<
```

```
      boost::function_types::parameter_types<f_type>, 1>::type
    arg1_type;

  typedef typename
    boost::mpl::at_c<
      boost::function_types::parameter_types<f_type>, 2>::type
    arg2_type;

  __WARNING< boost::is_same<
      typename boost::remove_const<
        typename boost::remove_reference<arg1_type>::type>::type,
      typename Fun::first_argument_type>::value, Fun > w1;

  __WARNING< boost::is_same<
      typename boost::remove_const<
        typename boost::remove_reference<arg2_type>::type>::type,
      typename Fun::second_argument_type>::value, Fun > w2;
};

#define CHECK_BINARY_FUNCTOR(F) __check_binary_adaptability<F>();
```

This class template is similar to the previous one, but this one uses `first_argument_type` and `second_argument_type` set by `binary_function`.

With our framework adaptable functors can be written safer because if the base class does not suit to the definition of `operator()` compilation warning is generated. The only limitation is the user of the framework has to start the verification manually. Our future work is to eliminate this limitation.

5. Conclusion

STL is widely-used standard C++ library based on the generic programming paradigm. STL increases efficacy of C++ programmers mightily because it consists of expedient containers and algorithms. On the other hand, improper application of the library results in undefined or strange behaviour.

Functors play an important role in the STL because they enable to execute user-defined code snippets in the library without significant overhead.

In this paper we detail a typical approach that results in a quite incomprehensible behaviour based on the functors' requirements. We show the background of the problem and argue for a non-intrusive approach as a plausible solution. Our solution has minimal overhead at runtime, but makes the usage of functors much safer.

Our framework also includes utilities that make the development of adaptable functors safer. Our approach is able to detect if the base class of the functor does not suit the functor at compilation time. Compilation warning is emitted, if a mistake is detected.

Acknowledgement

This research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

References

[1] A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001.
[2] ANSI/ISO C++ Committee. Programming Languages – C++. ISO/IEC 14882:1998(E). American National Standards Institute, 1998.
[3] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library,* Addison-Wesley, 1998.
[4] M. H. Austern, R. A. Towle, A. A. Stepanov, *Range partition adaptors: a mechanism for parallelizing STL*, in ACM SIGAPP Applied Computing Review 1996 **4(1)**, pp. 5–6,
[5] T. Becker, *STL & generic programming: writing your own iterators*, C/C++ Users Journal 2001 **19(8)**, pp. 51–57.
[6] M. Biczó, K. Pócza, Z. Porkoláb, I. Forgács, *A new concept of effective regression test generation in a C++ specific environment*, In Acta Cybern., **18** (2008), pp. 481–512.
[7] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications,* Addison-Wesley, 2000.
[8] G. Dévai, N. Pataki, *A tool for formally specifying the C++ Standard Template Library*, In Ann. Univ. Sci. Budapest., Comput. **31**, pp. 147–166.
[9] G. Dévai, N. Pataki, *Towards verified usage of the C++ Standard Template Library*, In Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST) 2007, pp. 360–371.
[10] D. Gregor, S. Schupp, *Stllint: lifting static checking from languages to libraries*, Software - Practice & Experience, **36(3)** (2006) 225-254.
[11] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, *Concepts: linguistic support for generic programming in C++*, in Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006), pp. 291–310.
[12] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, *Algorithm specialization in generic programming: challenges of constrained generics in C++*, in Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2006), pp. 272–282.
[13] B. Karlsson: *Beyond the C++ Standard Library: An Introduction to Boost,* Addison-Wesley, 2005.
[14] S. Meyers, *Effective STL*, Addison-Wesley, 2003.
[15] D. R. Musser, A. A. Stepanov, *Generic Programming*, in Proc. of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation, Lecture Notes in Comput. Sci., **358** 1988, pp. 13–25.

[16] N. Pataki, *C++ Standard Template Library by Safe Functors*, in Proc. of The 8-th Joint Conference on Mathematics and Computer Science, Selected Papers, pp. 357–368.
[17] N. Pataki, *Testing by C++ Template Metaprograms*, in Acta Univ. Sapientiae, Inform., **2**(2), pp. 154–167.
[18] N. Pataki, Z. Porkoláb, Z. Istenes, *Towards Soundness Examination of the C++ Standard Template Library*, In Proc. of Electronic Computers and Informatics, ECI 2006, pp. 186–191.
[19] N. Pataki, Z. Szűgyi, G. Dévai, *C++ Standard Template Library in a Safer Way*, In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.
[20] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, *Runtime Concepts for the C++ Standard Template Library*, In Proc. of the 2008 ACM Symposium on Applied Computing, pp. 171–177.
[21] D. Rayside, Z. Benjamin, R. Singh, J. P. Near, A. Milicevic, D. Jackson, *Equality and hashing for (almost) free: Generating implementations from abstraction functions*, In Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE 2009) pp. 342–352.
[22] Z. Porkoláb, *Functional Programming with C++ Template Metaprograms* in Proc. of Central European Functional Programming School, Revised Selected Lectures, Lecture Notes in Comput. Sci., **6299**, pp. 306–353.
[23] Z. Porkoláb, Á. Sipos, N. Pataki, *Inconsistencies of Metrics in C++ Standard Template Library*, In Proc. of 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2007, Berlin, pp. 2–6.
[24] B. Stroustrup, *The C++ Programming Language - Special Edition*, Addison-Wesley, 2000.
[25] Z. Szűgyi, Á. Sinkovics, N. Pataki, Z. Porkoláb, *C++ Metastring Library and its Applications*, In Proc. of Generative and Transformational Techniques in Software Engineering 2009, Lecture Notes in Comput. Sci., **6491**, pp. 461–480.
[26] M. Torgersen, *The Expression Problem Revisited – Four New Solutions Using Generics*, in Proc. of European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Comput. Sci., **3086**, pp. 123–143.
[27] L. Zolman, *An STL message decryptor for visual C++*, In C/C++ Users Journal, 2001 **19(7)**, pp. 24–30.
[28] I. Zólyomi, Z. Porkoláb, *Towards a General Template Introspection Library*, in Proc. of Generative Programming and Component Engineering: Third International Conference (GPCE 2004), Lecture Notes in Comput. Sci., **3286**, pp. 266–282.

Dept. of Programming Languages and Compilers, Fac. of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary

E-mail address: patakino@elte.hu