On the Validated Usage of the C++ Standard Template Library

Bence Babati
Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
Hungary, Budapest
babati@caesar.elte.hu

Gábor Horváth
Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
Hungary, Budapest
xazax@caesar.elte.hu

Norbert Pataki
Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
Hungary, Budapest
patakino@elte.hu

Attila Páter-Részeg
Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
Hungary, Budapest
athilarex@caesar.elte.hu

ABSTRACT

The C++ Standard Template Library (STL) is the most well-known and widely used library that is based on the generic programming paradigm. The STL takes advantage of C++ templates, so it is an extensible, effective but flexible system. Professional C++ programs cannot miss the usage of the STL because it increases quality, maintainability, understandability and efficacy of the code.

However, the usage of C++ STL does not guarantee error-free code. Contrarily, incorrect application of the library may introduce new types of problems. Unfortunately, there is still a large number of properties that are tested neither at compilation-time nor at runtime. It is not surprising that in implementation of C++ programs so many STL-related bugs may occur.

It is clearly seen that the compilation validation is not sophisticated enough to detect subtle misusages of this library. In this paper, we propose different approaches for the validation of the C++ STL's usage. We take advantage of aspect-oriented programming paradigm, metaprogramming techniques, static analysis, and automated debugging tool as well. We compare the presented approaches comprehensively, the pros and cons of these techniques are considered, evaluated and highlighted.

CCS CONCEPTS

• **Software and its engineering** → *Software verification*; *Automated static analysis*; *Dynamic analysis.*

KEYWORDS

C++, Standard Template Library, metaprogamming; static and dynamic analysis; aspect-oriented programming

ACM Reference Format:
Bence Babati, Gábor Horváth, Norbert Pataki, and Attila Páter-Részeg. 2019. On the Validated Usage of the C++ Standard Template Library. In *9th Balkan Conference in Informatics (BCI'19), September 26–28, 2019, Sofia,Bulgaria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3351556.3351570

BCI'19, September 26–28, 2019, Sofia,Bulgaria

ACM ISBN 978-1-4503-7193-3/19/09…$15.00
https://doi.org/10.1145/3351556.3351570

1 INTRODUCTION

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [3]. In this way, containers are defined as class templates and many algorithms can be implemented as function templates [2]. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [26]. C++ STL is widely-used because it is a very handy, standard library that provides beneficial containers (like list, vector, map, etc.), a lot of algorithms (like sort, find, count, etc.) among other utilities.

The STL was designed to be extensible [6]. 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. The expression problem is solved with this approach [27]. STL also provides adaptor types which transform standard elements of the library for a different functionality [18]. By design, STL is implemented with application of C++ templates to ensure the efficiency. A runtime model of this approach is available [22].

However, the usage of C++ STL does not guarantee error-free code [8]. Contrarily, incorrect application of the library may introduce new types of problems [5].

One of the problems is that the error diagnostics are usually complex and very hard to figure out the root cause of a program error [28, 29]. Violating requirement of special preconditions (e.g. sorted ranges) is not checked, but results in runtime bugs [21]. 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*. Further reference of invalid iterators causes undefined behavior [7].

Another common mistake is related to algorithms which are deleting elements. The STL 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 one needs to invoke a specific erase member function to remove the elements

physically. Therefore, for example, the `remove` and `unique` algorithms do not actually remove any element from a container [15]. The C++17 standard provides the `[[nodiscard]]` attribute to emit warning in similar cases, but many large code bases cannot use C++17 standard-compliant compilers. Moreover, this attribute is not used comprehensively in the library.

The aforementioned `unique` algorithm has uncommon precondition [9]. Equal elements should be in consecutive groups. In general case, using `sort` algorithm is advised to be called before the invocation of `unique`. However, `unique` results in a well-defined behavior but its result may be counter-intuitive at first time.

Some of the properties are checked at compilation time. For example, the code does not compile whether one uses `sort` algorithm with the standard list container because the list's iterators do not offer random accessibility [20]. Other properties are checked at execution time. For example, the standard vector container provides an `at` method which tests whether the index is valid and it raises an exception otherwise.

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

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`) are typically used with functors because of efficiency. These containers and algorithms need *strict weak ordering* [19]. Containers become inconsistent, if the used functors do not meet the requirement of strict weak ordering [16].

Certain containers provide member functions with the same names as STL algorithms. This phenomenon has many different reasons, for instance, efficiency, safety, or riddance of compilation errors to name a few. For example, as mentioned, list's iterators cannot be passed to `sort` algorithm, hence code cannot be compiled. To overcome this problem list has a member function called `sort`. The list container type also provides `unique` method. In these cases, although the code compiles, the calls of member functions are preferred to the usage of generic algorithms because of the runtime performance.

In this paper, we present approaches that analyze the usage of STL because it is clearly seen that only the compilation validation is not enough to realize the possible misusages. We take advantage of different techniques: template metaprogramming [23], static analysis, debugger-based non-intrusive solution and an aspect-oriented method [12]. The metaprogramming approach uses standard C++ constructs and compilers. The static code analysis tool utilizes the Clang architecture [13]. This approach matches patterns on abstract syntax trees (AST) with predicates written in a functional style. The debugger-based method takes advantage of the `gdb` debugger. Aspect-oriented verification is a known method in dynamic analysis [24], our solution deals with AspectC++ [25].

The rest of this paper is organized as follows. In section 2, we propose some approaches for validating the usage of STL. We evaluate these methods and compare them in section 3. Finally, this paper concludes in section 4.

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

Figure 1: Function template for warning emission

```
struct COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR
{
};
```

Figure 2: Dummy type for warning emission

```
warning( COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR() );
```

Figure 3: Instantion of template to emit specific warning

```
warning C4100: 't' : unreferenced formal parameter
...
see reference to function template instantiation 'void
warning<COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR>(T)'
being compiled

        with
        [
            T=
COPY_ALGORITHM_WITHOUT_INSERTER_ITERATOR
        ]
```

Figure 4: Emitted compilation warning by Microsoft Visual Studion

2 APPROACHES

2.1 Metaprogramming

This approach is based on the template construct of C++ which is able to evaluate Turing-complete checks at compilation time [23]. However, this approach cannot deal with abstract syntax trees, only template instantiations can be considered.

The first goal is to emit custom warning messages from the compilers even whether the compiler is not open source. C++11 provides static assert for emit compilation errors in specific cases. However, there are many cases when detected code snippets do not cause any runtime problems. We highlight possible problems, but do not break the compilation process because large code bases still shall be compiled. Figure 1 presents the function template for generating customized warning.

This template generates warning when it is instantiated and its template argument is highlighted in the warning diagnostics, so new kind of warning requires a dummy type. A typical example can be seen in Figure 2.

One can instantiate the template function with this dummy type in a straightforward way. It can be seen in Figure 3.

Different compilers emit the warning in different ways. Microsoft Visual Studio presents the warning message that can be seen in Figure 4.

The compilation warning emitted by the g++ compiler can be seen in Figure 5.

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

Figure 5: Emitted compilation warning by g++

With this approach, we can modify our STL implementation to evaluate specific instantiations: we have to add the warning method to `vector<bool>` specialization's constructor or to the `auto_ptr` partial specializations. The conversion of reverse iterators also can be detected with this technique [17]. There are some cases when more sophisticated approach is required. The `iterator_traits` type also can be extended with extraordinary member types to pass metainformation from the containers to the algorithms (e.g. invariant sortedness of container). The metaprogramming facilities also can use this metadata and emit compilation warnings in specific cases, e.g. `count` algorithm on a sorted container [19]. Typically programmers may use believe-me marks to disable specific warnings [19].

2.2 Static analysis

During static analysis, we reason about a program without executing it. Our approach is based on pattern matching on the abstract syntax tree (AST) of the analyzed program source code. The AST provides sufficient amount of information to answer several questions regarding the source code. Clang [14] is a widely used tool in static analysis [4]. Our approach also utilizies Clang compiler infrastructure [9].

However, in order to parse the source of the program we need to know the exact compiler arguments that were used to compile that application. This is necessary because the compiler arguments can modify the semantics of the source code, for example, macros can be defined using compiler arguments.

To collect the compilation arguments, the most robust and portable way is to use fake compilers that are logging their parameters and forwards them to a real compiler afterwards. This way, the logging itself is independent of the make system that is used. The source code is parsed with the same compilation parameters that were logged. When the user compiles for multiple platforms it is often advised to run the static analysis for each of them.

After we retrieved the AST of the analyzed program, the pattern matching process begins. Multiple patterns are matched lazily on the AST with only one traversal. The source positions for the matched nodes in the AST are collected.

The source positions in the collected results are filtered based on exclude lists that contain the false positive matches. These exclude lists have to be maintained by the user of the tool. Afterwards, the positions are translated into user-friendly warning messages to emit.

One of the downsides is that the compiler can only parse one translation unit at a time. Some useful information might reside in a separate translation unit making it impossible to detect some class of issues. Fortunately, due to the structure of STL most of the library code is available in system header files. For this reason, when a translation unit is utilizing some STL features, the corresponding headers are likely to be the part of that unit. This structure mitigates the limitations of the compiler, translation unit boundaries are not likely to be a problem when analyzing STL misuse patterns [10].

```
MatcherProxy StlEmptyCheckPred::getMatcher()
{
  DeclarationMatcher container = unless( anything() );
  for ( const auto & e : gContainers )
    container = anyOf( recordDecl( hasName( e ) ),
                       container );

  return
    memberCallExpr(
      on( hasType( container ) ),
      callee( methodDecl( hasName( "size" ) ) ),
      unless( anyOf(
          hasAncestor( binaryOperator(
            unless( has( integerLiteral(
              equals( 0 ) ) ) ),
            unless( anyOf(
              hasOperatorName( "&&" ),
              hasOperatorName( "||" ) ) ) ) ) ),
          hasAncestor( varDecl() ) ) ) ).bind( "id" );
}
```

Figure 6: Source code of the matcher that checks whether a container's size is compared to 0

```
std::list<int> s;
//...
if ( s.size() > 0 )
{
  std::list<int>::iterator it = s.begin();
  // ...
}
```

Figure 7: Source code of the matcher that checks whether a container's size is compared to 0

Each of the checks that are able to detect a certain class of bad smells is implemented as a predicate on the AST. These predicates are loosely coupled. We have focused on the extensibility, thus it is convenient to add further checkers to our tool.

An example can be seen in the Figure 6. This check aims to validate whether someone queries the container's emptiness the following way:

The code above that can be seen in Figure 7 is straightforward and correct, but such use-cases can cause unnecessary runtime overhead using `size` compared to `empty` method [15]. The code in Figure 6 uses the `gContainers` variable that contains the list of the containers in the STL. The implementation of the check is written in a functional style.

This approach is more subtle than metaprogramming and it is able to detect bugs and smells that cannot be discovered by metaprograms, for example: using the "swap-trick" to decrease a vector's

```
define checkinterval
  if $argc == 2
    set $first = $arg0
    set $last = $arg1
    set $l = 1
    while $first._M_current < ($last._M_current - 1) &&
          $l == 1
      set $act = $first
      set ++$first._M_current
      if  *($first._M_current) < *($act._M_current)
        set $l = 0
      end
    end
    if $l == 1
      printf "PASSED\n"
    else
      printf "ERROR, interval not sorted\n"
    end
  end
end
```

Figure 8: Source code of the debugger script that checks whether an interval is sorted

unnecessary allocated capacity, one can use `shrink_to_fit` convenience method since C++11 [9]. It can also be detected whether someone calls an algorithm that removes element(s) from a container but forgot to call the container's `erase` method to actually remove the element(s). This tool also can detect instantiations of `vector<bool>` or constructions of containers which hold `auto_ptr` objects (COAPs) because using COAP objects is very error-prone. Most of the checks we developed were open sourced and contributed to the Clang Tidy [1] toolset. Moreover, our tool is in-use at our industrial partner, so this approach is validated with large code bases.

2.3 Debugger-based runtime validation

This approach is a non-intrusive, runtime method for validating the usage of STL. This approach is based on the `gdb` debugger tool. We set breakpoints to specific locations in the library and when the execution is suspended we can evaluate the intervals, iterators, the state of the containers, etc. After evaluation, we can continue the execution or indicate an error. Correct and convenient usage of gdb requires compilation with debug mode enabled. In this case, debug information makes the application significantly slower.

We have developed gdb scripts for validating the usage of STL [11]. For instance, the 8 presents our `checkinterval` script that validates whether an interval is sorted based on `operator<`.

We can set breakpoints to our specific library implementation, for instance, in gdb, Figure 9 presents how we can define two breakpoints that suspend the execution when one of the overloaded `unique` algorithm is invoked:

After evaluation at specific points of the STL implementation, one can continue the execution. However, the gdb scripts are not sophisticated properly, so we have developed an automation tool. This tool has many goals:

```
break /usr/include/c++/5.3.1/bits/stl_algo.h:992
break /usr/include/c++/5.3.1/bits/stl_algo.h:1022
```

Figure 9: Configuration of breakpoints

- Preparation: launches the debugger with logging and added breakpoints
- Data processing: logs the debugger's output and process it. The important data must be filtered and forward it to the analysing component.
- Administration: this component maintains the state of objects because the STL objects may change between two breakpoints. For instance:
  - Iterator administration: keeps iterators' data update: name, validity, referred memory address.
  - Container administration: keeps containers' state update: type, capacity, size, iterators etc.
- Analysis: this component evaluates the maintained objects in an iterative way. It drives the debugger, the data processing component and passes input for the logging component and finally makes the gdb to continue the execution.
- Logging: the framework returns a comprehensive list of every critical STL-related code snippet. Every item of this list appears in a logfile and describes whether the test fails, passes or warns a potentially incorrect usage.

The tool detects problems related to algorithms: using `find` and `count` algorithms on sorted containers, using algorithms with special precondition (e.g. `lower_bound` that needs sorted interval but this requirement is not validated at compilation time nor at runtime). Also, incorrect usages of copying and removing algorithms can be realized. This tool is able to detect the invalidation and the conversion of iterators, as well. Moreover, special instantiations of containers (e.g. COAP and `vector<bool>`) are discovered.

2.4 Aspect-oriented approach

This is a runtime validation method for STL usage. The main idea behind this technique is that extend or modify the original program runtime behavior without changing the source code. It usually is done by adding extra source code just before the compilation. This extra source code is woven into original source code, but it is not changed physically.

There are many implementations which can provide these features for different languages. For this project, the AspectC++ was used, which is a solution for aspect-oriented programming in C++. Let us see briefly how it works. This approach extends the C++ language with some constructs which can be used to define join points, the relations between them and so on. There are many points which can woven in the source code, just for instance, one can wrap function calls or extend classes. Also, the standardized C++ language can be used to implement the changed behavior of these points. However, a non-standard compiler (aspect weaver) must be used that generates C++ code that can be compiled with standard C++ compilers.

For example, let us change the original behavior in order to print a message "function called" at each `foobar` function calls. The source code is straightforward, it defines a function called `foobar`

```
#include <iostream>

int foobar()
{
  return 42;
}

int main()
{
  std::cout << foobar() << std::endl;
}
```

Figure 10: Implementation of the example functions

```
#include <iostream>

aspect Example
{
  pointcut foobar_pc = "int foobar()";

  advice execution(foobar_pc()) : after()
  {
    std::cout << "function called" << std::endl;
  }
};
```

Figure 11: AspectC's aspect example

and it is called in the `main` function. The implementation of `foobar` and `main` functions can be seen in Figure 10.

Without any modifications, it just prints "42" on standard output. An aspect-oriented code snippet can be seen in Figure 11. After applying this code using AspectC++, it prints an extra line "function called" after "42" on standard output.

The pointcut defines the function signature pattern, which needs to be matched at function execution. If it does meet the requirement, the body of the advice will be called after the original function body and it will print out the given message.

Under the hood, the waiving is done before the compilation. The AspectC++ transforms the original source and the defined aspects into a new source code which is standard C++ code and it can be compiled with C++ any compiler. For source code transformation, it uses Clang which can do AST-level source code changes.

As it can be seen, aspect-oriented approach can be used for wide range of things, STL usage validation is one of them. The rules usually are kind of items which needs to be fulfilled at each given usage, e.g. using `find` algorithm on sorted associated containers. Since this approach is a runtime methods, the validations become straightforward. The code of validation reaches the containers and iterator to supervise. Probably this is the main advantage of this method against other compile time approaches, for example, we exactly know the size of the accessed container, because we can access it in the memory. So aspect-oriented approach can be used in an essential way, let us see two examples for them.

The first of these examples is practical to depict the methodology behind this paradigm. The `unique` algorithm can be used to

```
std::vector<int> numbers { 1, 1, 2, 3, 2, 4 };
auto uniqueEnd = std::unique(numbers.begin(),
                             numbers.end());
std::for_each(numbers.begin(),
              uniqueEnd,
              [] (int x) { std::cout << x; });
// prints: 1 2 3 2 4
```

Figure 12: Potential misuse of the `unique` algorithm

```
pointcut usageUnique() = "% std::unique<...>(...)";

advice call(usageUnique()) : after()
{
  if ( !isUnique(*tjp->arg<0>(), *tjp->result() ) )
  {
    // report the problem
  }
}
```

Figure 13: Validation of the `unique` algorithm

```
main.cc:7: error: std::unique is used on improper range
```

Figure 14: Warning emission at runtime

remove the same consecutive elements from a container. However, as mentioned before, it does not remove them from the container, just move them to the end and gives a new end iterator. This is the position where the unnecessary elements start. However, if the targeted container is not sorted, the `unique` cannot do what is expected naturally. The elements will not be reordered in this case as it is probably expected. In Figure 12, the value 2 will be printed twice because they were not next to each other, the container was unprepared for the `unique`.

This issue can be caught with aspect-oriented programs. The `unique` algorithm calls need to be checked whether the resulted range has element duplication. The iterators are accessible at the point of function calls and exactly the same what the `unique` sees. The core of our implementation can be seen in Figure 13. In the code snippet, we put a checkpoint a `unique` function calls and analyze the resulted range, from begin iterator to the resulted iterator. The parameters of the function and its result can be accessed through `tjp` object which refers to the currently analyzed context. In this example, the `*tjp->arg<0>()` is the first parameter of `unique` which is the begin iterator and the `*tjp->result()` is the resulted iterator. The returned iterator is available after the original function is evaluated. The `isUnique` function does the duplication checking, but its implementation is not so relevant in this example.

When the framework has found some issues, diagnostics are printed out on standard error in the same format as the compilers do, for instance, Figure 14 presents a warning diagnostics. Since the runtime objects in the memory are accessible, the program would not report any false positives.

The previously shown erroneous usage of the `unique` algorithm is one of the most straightforward case to find because only one

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> v { 1 };
  auto it = v.begin();
  for( int i = 1; i <= 1000; ++i )
  {
    v.push_back( i * 2  );
  }  // vector reallocates
  std::cout << *it; // it is invalid here
}
```

Figure 15: An example for iterator invalidation

evaluation is required in certain point of execution. However, during the STL usage there can be more difficult problems to find, like iterator invalidation. Just to introduce the original problem, iterator invalidation can happen in many cases, so just let us see a `vector`-related one. After a new element is inserted to the end of a `vector`, the already constructed iterators may become invalid:

- end iterator is always invalidated
- all iterators are invalidated if the underlying memory chunk needs to be reallocated when the container's size exceeds its capacity

This problem requires more sophisticated approach, so in order to analyze it, the already constructed iterators for the given container are needed to taken care of.

There are three parts of this analysis in our implementation. First of all, the iterator constructions are necessary to be checked, there are some member functions which can construct new iterators like `begin()`. All of the constructed iterators are watched for each container till they go out of scope.

Second of all, the containers need to be watched whether they have been changed. In case of the `vector` insertion example, the given two cases need to be handled properly and mark the affected iterators invalid in the following.

Thereafter, if the container's elements get accessed through an iterator, it is required to check whether it is valid. In case of invalidated iterator (for instance Figure 15), the program can print an error message just as before.

With this approach, we have implemented various validations, for instance, invocation of `find` algorithm on sorted associative containers, usage of invalidated iterators, correct usage of `copy` and `transform` algorithms. Usage of `vector<bool>` can be detected, as well. Application of stateful functors is also realized by our approach.

3 EVALUATION

For the evaluation of methods, we consider the following properties: sophistication, intrusiveness, STL implementation specificity, applicability and general performance. We also define whether a method has some specific features or requirements.

Sophistication is an essential question regarding how many problems can be detected by the approach. Intrusiveness defines whether a solution requires changes in the implementation of the STL. Non-intrusive approaches are preferred to intrusive methods. Many STL implementations are available (e.g. STLport, Dinkumware STL, HP STL, SGI STL), so preferred methods do not need to know the actual implementation of the library. STL implementation-specific solutions are more complicated to port to a different platform. STL-based code can be found in various software artifacts, for instance, standalone applications (with an entry point), compiled libraries, header-only template libraries. Applicability defines what kinds of artifacts are supported with the proposed methods. Actual performance issues cannot be compared correctly because some approaches work at runtime and other approaches that validate at compilation time. However, we can compare the general performance, as well.

Obviously, the sophistication depends on the fact whether an approach runs at compilation time or runtime because many details are known only at runtime (e.g. actual size of a container, etc.). Runtime approaches can detect more problems. However, special contexts are easier to analyze with static analysis (e.g. size is equal to 0) and there are problems that can be detected only with static analysis (e.g. include dependencies [5]). The metaprogramming technique is not a straightforward solution because its capabilities are quite limited. Static analysis is more sophisticated than metaprogramming. Our static analysis approach cannot reason about runtime information, but a symbolic execution engine may improve our validations later.

We have implemented various validations with gdb-based and aspect-oriented approaches. Table 1 presents the implemented(+) and unimplemented(-) validations. We have implemented more validation based on the gdb, so this technique is considered slightly more sophisticated.

The metaprogramming approach is intrusive, modification of the STL implementation is required. The static analysis and the debugger-based are non-intrusive techniques. Aspect-orientation has been developed as non-intrusive approach, so the validation technique using aspect-oriented constructs is also non-intrusive.

Since metaprogramming technique is intrusive, this approach is STL implementation-specific. The static analysis does not deal with library-specific solutions, the checkers are implemented regarding the C++ standard. Unfortunately, the debugger-oriented validation is implementation-specific because gdb requires the line number of the source code in the library, it cannot deal with overloaded templates only with their names. The aspect-oriented approach can be used with many STL implementations because it is not library-specific.

Template instantiation triggers the evaluation of metaprograms' checkers. Therefore, this approach cannot be used in template (header-only) libraries. Metaprograms support applications and libraries that are compiled actually. Static analysis is the best solution regarding applicatability because it can be used with source code, it does not matter whether it is a code of a library or an application. The debugger-based approach requires an entry point for execution, so libraries are not supported. It can deal with applications exclusively. The aspect-oriented solution patches the library with runtime validations, it does not matter whether the invocation

Table 1: Comparison of implemented validations with runtime approaches

Validation	gdb-based	AOP
Valid ranges for input	+	-
Iterator invalidation	+	+
Conversion of iterators	+	-
Usage of `vector<bool>`	+	+
Usage of COAP	+	-
Proper usage of copy-like algorithms	+	+
Proper usage of remove-like algorithms	+	-
Special preconditions (like sorted input intervals)	+	-
Using `find` or `copy` algorithms on associative containers	+	+
Stateful predicates	-	+
Precondition of `unique` algorithm	+	+
Strict-weak ordering	-	+

of STL functionalities comes from a new library of application. However, the validations evaluated at runtime, so executable, running application can be considered technically.

Regarding the performance, the metaprogramming approach is the best. It is evaluated during the usual compilation process with some extraordinary template instantiations with no runtime overhead. The static analysis runs separately from the compilation, so it requires a new build process which makes the approach slow, but fortunately, it has no runtime overhead. Using debug mode for compilation and execution means significant runtime overhead. Regarding the performance, using the aspect-oriented approach is better. The aspect-oriented solution uses static analysis technique to find the "breakpoints" that performs much better than gdb.

Runtime approaches do not generate false positive and true negative results, these tools report only when the problem actually occurs. The compile time techniques (metaprogramming and static analysis) may find problematic points in the code that is not bug actually. The metaprogramming facilities provide "Believe-me" marks to decrease the number of false positives. Runtime approaches are preferred during testing process because they cause runtime overhead, unnecessary log messages and may cause changes in the execution. Unfortunately, both approaches have major drawbacks regarding the applied technology. On one hand, non-standard, aspect-oriented language elements require the aspect weaver. On the other hand, debuggers are widely-used tools but they require debug mode compilation that results in signifact overhead at runtime, so these approaches should not be applied in production, especially the debugger-based solution.

Briefly, runtime approaches are able to detect more problems, but static analyses are not so far. Many STL-related problems can be implemented with a Clang-based, library-independent solution as well. However, it has significant runtime because it run as a standalone compilation tool with extraordinary validations and false positives may appear. It is suggested to use in nightly builds. The runtime solutions are preferred in staging. The pros and cons of the proposed approaches are in Table 2 and Table 3.

4 CONCLUSION

C++ STL is based on the generic programming approach. It is flexible, efficient, handy library that increases code quality significantly, but the recent compilers cannot guarantee comprehensively whether the library is used perfectly. There are many subtle misusages which require more specific knowledge. These semantically incorrect usages may affect the runtime correctness and performance or may cause undefined behavior and inconsistent containers.

In this paper, we proposed methods to detect subtle misusages of the library. We presented a technique briefly that takes advantage of the C++ templates and triggers the compiler to validate the usage of the STL. We argued for a Clang-based static analysis tool, as well. Two runtime approaches are also proposed. The first one takes advantage of the gdb debugger tool and the second one utilizies a non-standard aspect-oriented C++ extension.

We have evaluated the proposed approaches. Static analysis performs quite well compared to the runtime solutions. Our static analysis method does not depend on the actual STL implementation. Dynamic analyses can help to detect the actual problems at execution time, but they are not preferred in production.

REFERENCES

[1] [n. d.]. Clang Tidy, a Clang-based linting tool. https://clang.llvm.org/extra/clang-tidy/. [Online; accessed 28-February-2019].

[2] Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Matthew H. Austern. 1998. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Bence Babati, Gábor Horváth, Viktor Májer, and Norbert Pataki. 2017. Static Analysis Toolset with Clang. In *Proceedings of the 10th International Conference on Applied Informatics (ICAI 2017).* 23–29. https://doi.org/10.14794/ICAI.10.2017.23

[5] Bence Babati and Norbert Pataki. 2017. Analysis of Include Dependencies in C++ Source Code. In *Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems (Annals of Computer Science and Information Systems)*, M. Ganzha, L. Maciaszek, and M. Paprzycki (Eds.), Vol. 13. PTI, 149–156. https://doi.org/10.15439/2017F358

[6] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[7] Gergely Dévai and Norbert Pataki. 2007. Towards verified usage of the C++ Standard Template Library. In *Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST) 2007.* 360–371.

Table 2: Major advantages of the proposed approaches

Approach	Major advantages
Metaprogramming	Standard-compliant compilers, fast solution
Static analysis	No runtime overhead
Debugger-based validation	Straightforward implementation
Aspect-oriented approach	Validations can be turned on and off easily

Table 3: Major disadvantages of the proposed approaches

Approach	Major disadvantages
Metaprogramming	Only deals with template instantiations
Static analysis	New compilation process
Debugger-based validation	Compilation with debug mode results in significant runtime overhead
Aspect-oriented approach	Aspect weaver is required

[8] Gergely Dévai and Norbert Pataki. 2009. A tool for formally specifying the C++ Standard Template Library. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 31 (2009), 147–166.

[9] Gábor Horváth and Norbert Pataki. 2015. Clang matchers for verified usage of the C++ Standard Template Library. *Annales Mathematicae et Informaticae* 44 (2015), 99–109.

[10] Gábor Horváth and Norbert Pataki. 2016. Source Language Representation of Function Summaries in Static Analysis. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '16)*. ACM, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/3012408.3012414

[11] Gábor Horváth, Attila Páter-Részeg, and Norbert Pataki. 2017. Detecting Misusages of the C++ Standard Template Library. In *Proceedings of the 10th International Conference on Applied Informatics (ICAI 2017)*. 129–136. https://doi.org/10.14794/ICAI.10.2017.129

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[13] Chris Lattner. 2008. LLVM and Clang: Next Generation Compiler Technology. Lecture at BSD Conference 2008.

[14] Bruno Cardoso Lopes and Rafael Auler. 2014. *Getting Started with LLVM Core Libraries*. Packt Publishing.

[15] Scott Meyers. 2001. *Effective STL*. Addison-Wesley.

[16] Norbert Pataki. 2011. Advanced Functor Framework for C++ Standard Template Library. *Studia Universitatis Babeş-Bolyai, Informatica* LVI, 1 (2011), 99–113.

[17] Norbert Pataki. 2012. Compile-time advances of the C++ Standard Template Library. *Annales Universitatis Scientiarium Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica* 36 (2012), 341–353.

[18] Norbert Pataki. 2012. Safe Iterator Framework for the C++ Standard Template Library. *Acta Electrotechnica et Informatica* 12, 1 (2012), 17–24. http://dx.doi.org/10.2478/v10198-012-0003-9

[19] Norbert Pataki and Zoltán Porkoláb. 2011. Extension of Iterator Traits in the C++ Standard Template Library. In *Proceedings of the Federated Conference on Computer Science and Information Systems*. IEEE Computer Society Press, Szczecin, Poland, 911–914.

[20] Norbert Pataki, Zalán Szűgyi, and Gergely Dévai. 2010. C++ Standard Template Library in a Safer Way. In *Proc. of Workshop on Generative Technologies 2010 (WGT 2010)*. 46–55.

[21] Norbert Pataki, Zalán Szűgyi, and Gergely Dévai. 2011. Measuring the overhead of C++ Standard Template Library safe variants. *Electronic Notes in Theoretical Computer Science* 264, 5 (2011), 71–83.

[22] Peter Pirkelbauer, Sean Parent, Mat Marcus, and Bjarne Stroustrup. 2008. Runtime Concepts for the C++ Standard Template Library. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*. ACM, New York, NY, USA, 171–177. https://doi.org/10.1145/1363686.1363734

[23] Zoltán Porkoláb. 2010. Functional Programming with C++ Template Metaprograms. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School (CEFP'09)*. Springer-Verlag, Berlin, Heidelberg, 306–353.

[24] Hiromasa Shin, Yusuke Endoh, and Yoshio Kataoka. 2007. ARVE: Aspect-Oriented Runtime Verification Environment. In *Runtime Verification*, Oleg Sokolsky and Serdar Taşıran (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–96.

[25] Olaf Spinczyk and Daniel Lohmann. 2007. The design and implementation of AspectC++. *Knowledge-Based Systems* 20, 7 (2007), 636–651. https://doi.org/10.1016/j.knosys.2007.05.004 Special Issue on Techniques to Produce Intelligent Secure Software.

[26] Bjarne Stroustrup. 2000. *The C++ Programming Language* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[27] Mads Torgersen. 2004. The Expression Problem Revisited – Four New Solutions Using Generics. In *ECOOP 2004 – Object-Oriented Programming (Lecture Notes in Comput. Sci.)*, Martin Odersky (Ed.), Vol. 3086. Springer Berlin Heidelberg, Berlin, Heidelberg, 123–146.

[28] Leor Zolman. 2001. An STL message decryptor for Visual C++. *C/C++ Users Journal* 19, 7 (2001), 24–30.

[29] István Zólyomi and Zoltán Porkoláb. 2004. Towards a General Template Introspection Library. In *Proc. of Generative Programming and Component Engineering: Third International Conference (GPCE 2004) (Lecture Notes in Comput. Sci.)*, Vol. 3286. 266–282.