Annales Mathematicae et Informaticae
Manuscript
http://www.ektf.hu/ami

Clang Matchers for Verified Usage of the C++ Standard Template Library*

Gábor Horváth, Norbert Pataki

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

25 January 2015

Abstract

The *C++ Standard Template Library (STL)* is the exemplar of generic libraries. Professional C++ programs cannot miss the usage of this standard library because it increases quality, maintainability, understandability and efficacy of the code. However, the usage of C++ STL does not guarantee bug-free or error-free code. Contrarily, incorrect application of the library may introduce new types of problems. Unfortunately, there is still a large number of properties are tested neither at compilation-time nor at run-time. It is not surprising that in implementation of C++ programs so many STL-related bugs occurred.

We match patterns on *abstract syntax trees* (AST) with the help of predicates. The predicates can be combined and define an embedded language. We have developed a tool which finds the potential missuses of the STL as a validation of our approach. The software takes advantage of the Clang ASTMatcher technology. The tool is in-use in the Ericsson. We advised new matchers that have get into the Clang code base.

Keywords: C++ STL, generic programming, Clang, AST, static analysis, code validation

MSC: 68N19 Other programming techniques

1. Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [2]. In this way containers are defined as class templates and

*This study/research was supported by a special contract No. 18370-8/2013/TUDPOL with the Ministry of Human Resources.

many algorithms can be implemented as function templates [1]. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [22]. C++ STL is widely-used because it is a very handy, standard library that contains 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 [3]. 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 [23]. STL also includes adaptor types which transform standard elements of the library for a different functionality [14]. By design, STL is implemented with application of C++ templates to ensure the efficiency. A runtime model of this approach is available [20].

However, the usage of C++ STL does not guarantee bug-free or error-free code [5]. Contrarily, incorrect application of the library may introduce new types of problems [10].

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 [24, 25]. Violating requirement of special preconditions (e.g. sorted ranges) is not checked, but results in runtime bugs [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*. Further reference of invalid iterators causes undefined behaviour [4].

Another common mistake is related to algorithms which are deleting elements. 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 physically. Therefore, for example the `remove` and `unique` algorithms do not actually remove any element from a container [9].

The aforementioned `unique` algorithm has uncommon precondition. 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` cannot result in an undefined behaviour 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 if one uses sort algorithm with the standard list container because the list's iterators do not offer random accessibility [18]. 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 [16].

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. Lint-like tools are based on static analysis for detect some kind of missuses that can be compiled but at runtime they cause problems.

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* [17]. Containers become inconsistent, if the used functors do not meet the requirement of strict weak ordering [12].

In this paper we argue for a new approach based on static analysis. Our approach matches patterns on abstract syntax trees with predicates written in functional style. We have created a tool as a validation of the approach. This tool detects many kind of missuses of the C++ STL in the source code. The tool uses the Clang architecture [8]. Our tool is utilized by our industrial partner.

This paper is organized as follows. In section 2 we analyze how our approach works, in section 3 the overview of our system is detailed. We briefly present our tool in section 4. We present the heart of some checkers as an example in section 5. We summarize other approaches that are based on static analysis in section 6. Finally, this paper concludes in section 7.

2. Pattern matching on syntax trees

Our approach is based on pattern matching on the syntax tree of the analyzed program source code. We use the syntax tree that is generated by the Clang [8] compiler. The syntax tree of a compiler contains sufficient amount of information to answer several questions regarding the source code.

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 efficient 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 was logged.

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

The source positions in the collected results are filtered based on exclude lists that contains of 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.

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 if 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.

The solutions presented in this paper are not utilizing any symbolic execution or other path sensitive data. Several patterns can be detected using only pattern matching on the AST and this pattern matching procedure is more efficient than symbolic execution.

3. Overview of the architecture

Each of the checkers that are able to detect certain class of bad smells are implemented as a predicate on the syntax tree. These predicates are loosely coupled. We focus on the extendibility, thus it is very easy to add further checkers to our tool.

Each predicate has two states representing whether they are activated in the current invocation of the tool or not and a list of matches (that was marked as false positives) that should be ignored. The predicates should not contain other states, because a new predicate object will be instantiated for each translation unit that is checked. There is no efficient way to preserve states between those invocations mainly because the user is allowed to invoke multiple instance of our checker tool on multiple translation units at the same time. If a predicate is activated it consists of a matcher and a callback. The matcher is an `ASTMatcher` object that is built by an embedded domain specific language available in the Clang ASTMatcher library. This is a first filter on the syntax tree and it can be retrieved by the `getMatcher` method. The callback can do further filtering to decide whether the match found by the ASTMatcher object is suitable to be reported to the user. The callback is the override of the `matches` method. The callback essential because while the matchers are extremely useful they can not express any possible patterns on the AST. The `getMatcher` method is never invoked on inactive checkers, the matcher expression will not be generated and the callback will never be called. The `MatcherProxy` returned by the `getMatcher` method is necessary because there are different kind of matchers for declarations, statements and types. The `MatcherProxy` is a discriminated union of fundamental matcher types from which the original type can be retrieved later.

```
struct Predicate {
  Predicate() : _active(false) {}
  virtual ~Predicate() {}

  virtual bool matches(const MatchResult &result_) = 0;
  virtual void configure(std::string conf_) = 0;

  virtual std::string getErrorMsg() = 0;
  virtual std::string getID() = 0;
  virtual std::string getBoundID() = 0;
  virtual MatcherProxy getMatcher() = 0;
  virtual std::vector<MatchPosition>& getExcludes() {
    return _excludes;
  }
  virtual bool isActive() { return _active; }
  virtual void setActive(bool active_) { _active =
      active_; }

protected:
  bool _active;
  std::vector<MatchPosition> _excludes;
};
```

Each predicate is configurable. This is necessary to make sure the predicates can adapt to wide variety of code bases. The ASTMatcher object and the callback function can make decisions based on configuration values. The predicates get only plain text as configuration because it is hard to predict what kind of configuration data is needed for future predicates. This gives some flexibility to the implementers. However, some basic tools are provided to parse key/value pairs.

Unfortunately static analysis tools are prone to false positive results. The users must be able to suppress the false positive warnings. The `_excludes` vector contains these locations. The exclude list is unique to all of the predicates. The exclude list is parsed after the configuration files was processed.

All of the predicates can be identified by a unique identifier. That identifier is used to determine if a predicate must be active in an invocation or not, what configuration values belong to that predicate, and which excludes should be added to its exclude list. The identifier can be gathered by invoking the `getID` method.

Every predicate can have a unique error message that is displayed when a match is found. This message is emitted for every source location that is detected by the given predicate.

The predicates can match not only a single node of the syntax tree but a subtree. In case a subtree is matched it is ambiguous which node should be used to retrieve the source location of the match. The `getBoundID` can be used to identify which node should be used as a source location to generate the warning report. For example if the subtree is a function call, there must be a way to determine what source location of the function call should be reported: the whole function call or one of its arguments.

```
Config::Config(BuildLogParser& log_parser) :
  _log_parser(log_parser) {
  ADD_MATCHER(StlBoolVectorPred);
  // ...
  ADD_MATCHER(StlPolimoContPred);
}
```

When a new checker is implemented it must be the subtype of the `Predicate` class. After all of the pure virtual methods are implemented the checker must be registered with a configuration class that handles the instantiation of the checker objects. The registration of the checker is done in the constructor of the configuration class. To avoid the necessity of modifying an unrelated file when implementing a new checker, the code that is responsible for registering checkers can be automatically generated by an external script that is inspecting the implementation files.

4. Matchers for STL usage validation

The architecture we developed proved to be useful. We implemented 14 checkers to detect STL specific errors or suspicious code snippets. We tried to focus on patterns that are most likely to appear in real world applications. These checkers are the following:

- Bool Vector checker warns about usages of vector of booleans. The reason is that `std::vector<bool>` is a template specialization that does not fulfil the requirements of `std::vector` [13].

- Container of `std::auto_ptr`s is a dangerous construct in C++. The `std::auto_ptr` is a smart pointer that manages its own resource by deleting the pointer it wraps in its destructor. The problem is that its copy constructor transfer the ownership of the pointer from the source of the copy to the target of the copy. When the user of the STL have a container that contains such pointers and use an algorithm on the container that involves copying then some pointers may reclaim their resources in an unintended way [13].

- Invoking the `std::find` and `std::count` algorithm on an associative container in STL is not efficient. The general `std::find` and `std::find` algorithms have no information about the internal representation of the containers. This is the reason why they cannot utilize that the objects in an associative container are ordered. The programmer should use the `find` and `count` methods of the associative container instead.

- One can get the wrapped iterator from a reverse iterator through the base method. However, using this method requires extra attention from the programmer, because the iterator retrieved this way is not pointing to the same object as the reverse iterator [15].

- The functors used with STL algorithms and containers should be derived from some specific STL types that adds some defines to the functor to make it possible to inspect the return type and the argument types of the functor. It is very error prone to define those types multiple times. This checker warns if the types do not match.

- Allocators should not have any state in C++98/C++03 codes.
- Functors used as predicates with STL algorithms should not have any state. This is important because it is undefined by the standard how many times may the functor object be copied.
- Functors are passed by value to algorithms. The polymorphism only works through pointers and references. Developers should not use virtual methods in functor classes and it can be the indicator of an error.
- The emptiness of a container should be checked with the `empty` method instead of using the `size` method. The empty method can be implemented more efficiently for most of the containers. Moreover more containers support the `empty` method than the `size` method. In a code base using `empty` for emptiness check the container types can be swapped more easily.
- The capacity and the number of elements in `std::vector` is not the same. The capacity defines how much memory the vector use and it can be much more than the required. To optimize the memory consumption it is a common trick to copy the contents of the vector to a temporary object and swap the vector with that temporary afterwards. Since the C++11 standard there is a much more comprehensible and better performing way to achieve the same result is using the `shrink_to_fit` method.
- Algorithms that remove elements from a container will not delete the elements from the container. They will only overwrite the elements that need to be removed with other elements from the end of the container. After running such algorithm at the and of the container there will be some redundant elements that needs to be erased using the `erase` method of the container.
- Copying algorithms cannot guarantee that the target container have sufficient size to store all of the elements that is in the source container. It is advised to use iterator adaptors for example the back inserter iterator adaptor to avoid buffer overflows.
- The containers in the STL are not designed to be the part of a class hierarchy. Using the containers in a polymorphic way is an error and should be avoided.
- Because of implicit type conversions a number can be assigned to a `std::string` object. In most cases this is a programming error and there is a missing explicit conversion to string.

Some of the bad smells and coding style advices that are found by our tool is not detected by other static analysis tools available at the time of writing this article. We found defects in the codebase of our industrial partner, however they did not provide us with any data on the number of defects they found.

5. Example

In this section we present two checkers' implementation briefly as an example. We focus on the technology and our approach.

Our tool analyzes if a programmer calls the `find` or `count` algorithm on a sorted container because this causes efficiency loss at runtime. The algorithm is a function template, thus it can called on the `set` or `multiset` object because the algorithm has a template parameter for the iterator. The heart of the checker takes advantage of Clang architecture. The following code snippet validates the usage of the `find` or `count` algorithm:

```
MatcherProxy StlFindCountPred::getMatcher() {
  return
    callExpr(
        callee(functionDecl(
        anyOf(hasName("std::find"),
              hasName("std::count")))),
      hasArgument(0,
        hasDescendant(expr(hasType(typedefType(hasDecl(
          matchesName(
            "std::(multiset|set).*"
            "::(const_iterator|iterator)"
           )))))))).bind("id");
}
```

This code checks if the parsed source code is a call expression, where the called function is one of enumerated standard algorithms and it is called on the sorted container. The code analyzes the type of algorithm's first argument. If the name of type matches to the arbitrary inner type `iterator` or `const_iterator` of `set` or `multiset` our tool reports warning to the user. However, this code is written in C++ but with functional approach.

The STL containers are not prepared to be used polymorphically. If a user decide to inherit from a container and cast a pointer to the derived type to a pointer to the container type it is probably the indicator of an issue.

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

  return
    implicitCastExpr(hasImplicitDestinationType(
      pointsTo(container))).bind("id");
}
```

The gContainers variable contains the list of the containers in the STL. The codesnippet above shows the power of creating matchers dynamically on the fly. We want to warn the user, if there is an implicit cast to a pointer to a container.

```
bool StlPolimoContPred::matches
  (const MatchFinder::MatchResult &result_)
{
  const ImplicitCastExpr* cast =
    result_.Nodes.getDeclAs<ImplicitCastExpr>("id");

  return cast->getCastKind () == CK_DerivedToBase;
}
```

There can be several types of casts, but we are only interested in those types of implicit casts, that involves derived to base conversion. There is no matcher to ensure the cast type. For this reason we have implemented a small callback to check the type of the cast.

6. Related work

There are some approaches to discover the erroneous STL usage with static analysis. STLlint was the first application that analyses source code for detecting inaccurate application of the library [7]. STLlint first parsed C++ code (with Edison Design Group C++ Front End [6]) and then transformed it into a simpler internal representation language called “Semple”. During this transformation process, STLlint replaced the implementation of STL components with simplified models that specify the interface (but not the implementation) of the components. The resulting program was then passed to the Semple static analysis engine that “executed” the program symbolically, checking that assertions (part of the component specifications) always prove true. Unfortunately, the support, maintenance and availability of this tool have been cancelled.

C++ template metaprogramming is an emerging new paradigm that is able to add further validation to the compiler [21]. This approach effectively can be used for evaluating the semantic usage of the STL. First, it is enough to parse the source code once. Second, this approach supports the extendibility of the library which is a major feature of the generic programming paradigm. However, this approach has disadvantages, as well. In the metaprogramming realm there is no AST or something high-level approach of the source code. Metaprogram developers usually deal with template instantiations to trigger compilation failures or warnings [17].

Some of the STL-related issues are detected in a metaprogram-driven way: overcome of stateful allocators and reverse iterators are implemented in [15], proper usage copying algorithms is verified in [17]. The usage of vector<bool> and containers of auto pointers can be detected [13]. The semantic check of functors are detailed in [11, 12].

7. Conclusion

We have developed a static analysis tool based on Clang technologies that is easy to extend. The success of the architecture is proven by the 14 checker that we implemented. Some of the checkers are implementing new guidelines that was not published nor checked before. During the development we contributed several patches to the Clang ASTMatcher library. The tool is utilized by our industrial partner.

References

[1] Alexandrescu, A.: "Modern C++ Design", Addison-Wesley (2001)

[2] Austern, M. H.: "Generic Programming and the STL: Using and Extending the C++ Standard Template Library", Addison-Wesley (1998)

[3] Czarnecki, K., Eisenecker, U. W.:"Generative Programming: Methods, Tools and Applications", Addison-Wesley (2000)

[4] Dévai, G., Pataki, N.: *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.

[5] Dévai, G., Pataki, N.: *A tool for formally specifying the C++ Standard Template Library*, Ann. Univ. Sci. Budapest. Comput., **31** (2009), pp. 147–166.

[6] Gibbs, T. H., Malloy, B. A., Power, J. F.: *Progression Toward Conformance for C++ Language Compilers*, Dr. Dobbs Journal **28(11)** (2003), pp. 54-60.

[7] Gregor, D., Schupp, S.: *STLlint: Lifting static checking from languages to libraries*, Software – Practice & Experience, **36(3)** (2006), pp. 225–254.

[8] Lattner, C.: *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.

[9] Meyers, S.: "Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library", Addison-Wesley (2001)

[10] Pataki, N.: *C++ Standard Template Library by Ranges*, in Proc. of the 8th International Conference on Applied Informatics (ICAI 2010), Volume 2, pp. 367–374.

[11] Pataki, N.: *C++ Standard Template Library by Safe Functors*, in Proc. of 8th Joint Conference on Mathematics and Computer Science, MaCS 2010, Selected Papers, pp. 363–374.

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

[13] Pataki, N.: *C++ Standard Template Library by template specialized containers*, Acta Universitatis Sapientiae, Informatica **3(2)** (2011), pp. 141–157.

[14] Pataki, N.: *Safe Iterator Framework for the C++ Standard Template Library*, Acta Electrotechnica et Informatica, Vol. **12(1)** (2012), pp. 17–24.

[15] Pataki, N.: *Compile-time Advances of the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **36** (2012), Selected papers of 9th Joint Conference on Mathematics and Computer Science MaCS 2012, pp. 341–353.

[16] Pataki, N., Porkoláb, Z., Istenes, Z.: *Towards Soundness Examination of the C++ Standard Template Library*, in Proc. of Electronic Computers and Informatics, ECI 2006, pp. 186–191.

[17] Pataki, N., Porkoláb, Z.: *Extension of Iterator Traits in the C++ Standard Template Library*, in Proc. of the Federated Conference on Computer Science and Information Systems, FedCSIS 2010, pp. 911–914.

[18] Pataki, N., Szűgyi, Z., Dévai, G.: *C++ Standard Template Library in a Safer Way*, In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.

[19] Pataki, N., Szűgyi, Z., Dévai, G.: *Measuring the Overhead of C++ Standard Template Library Safe Variants*, Electronic Notes in Theoret. Comput. Sci., **264(5)** (2011), pp. 71–83.

[20] Pirkelbauer, P., Parent, S., Marcus, M., Stroustrup, B.: *Runtime Concepts for the C++ Standard Template Library*, in Proc. of the 2008 ACM symposium on Applied computing, pp. 171–177.

[21] Porkoláb, Z.: *Functional Programming with C++ Template Metaprograms*, Lecture Notes in Comput. Sci., **6299** (2010), pp. 306–353.

[22] Stroustrup, B.: “The C++ Programming Language”, Addison-Wesley (1999)

[23] Torgensen, M.: *The Expression Problem Revisited – Four new solutions using generics*, Lecture Notes in Comput. Sci., **3286** (2004), pp. 123–143.

[24] Zolman, L.,: *An STL message decryptor for visual C++*, C/C++ Users Journal, **19(7)** (2001), pp. 24–30.

[25] Zólyomi, I., Porkoláb, Z.: *Towards a General Template Introspection Library*, Lecture Notes in Comput. Sci., **3286** (2004), 266–282.

Gábor Horváth, Norbert Pataki
Department of Programming Languages and Compilers,
Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail:
`xazax.hun@gmail.com, patakino@elte.hu`