Correctness of Generative Programs

Theses of the PhD Dissertation
2013

Norbert Pataki
patakino@elte.hu

Supervisor: **Dr. Zoltán Porkoláb, associate professor**
Eötvös Loránd University, Department of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C

Eötvös Loránd University, Doctoral School of Informatics
Doctoral Program: Basics and Methodology of Informatics
Head of School: Dr. András Benczúr
Head of Program: Dr. János Demetrovics

---

1 Introduction

The *C++ Standard Template Library (STL)* is the exemplar of generic libraries [1]. It is one of the C++ standard libraries. Professional C++ programs cannot miss the usage of the STL because it increases quality, maintinability, understandability and efficacy of the code [20].

However, the usage of C++ STL does not guarantee bugfree 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 suprising that in implementation of C++ programs so many STL-related bugs occured [5]. The overcome of these problems is in the centre of my researches with saving the flexibility and efficacy of the STL.

2 C++ Standard Template Library

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. The basic components of the STL are: containers, algorithms, iterators, functors, adaptors and allocators.

The basic task of the *containers* (e.g. `vector`, `set`, `map`, etc.) is to locate the data in the memory, to store them and keeping the memory consistent. The *iterators* define the access of elements that are stored in the memory with a uniform interface. The *algorithms* are parametrized by the used iterators, thus these are container-independent templates for frequent tasks, such as searches, sorts, copy or count elements. The iterators guarantee the indepence of algorithms and containers. An important component of the STL is the *functor*. Functors enable the execution of user-defined code snippets in an effective way. 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. Allocators have been developed as an abstraction of memory models. Every standard container offers a template parameter for the customization of memory allocation. The last template parameter defines the type of the allocator that has default value but another one can be given.

3 Motivation

In my dissertation I show the difficulties that programmers have to overcome when the STL is in-use. I review the problems that are caused when the STL is misused. These problems can cause difficult message diagnostics, non-portable code, wrong results, memory leaks, corrupt or inconsistent data structures or unnecessary overhead. Some of the present problems can be overcome by my results, others require further researches.

In the dissertation I overview the problems that related to the algorithms: misuse of copying algorithms that results in runtime errors, the potential errors of the removing algorithms. I show why the usage of `unique` algorithm is counter-intuitive. I investigate the algorithms which have special preconditions that cannot be verified by the compilers. In addition, I examined the `count` and `find` algorithms, and I show environment in which these algorithms do not work correctly.

In the dissertation I show typical errors that related to containers: containers of `auto_pointer`s (COAPs), the cause of `vector<bool>` specialization and its counter-standard behaviour. I mention the problems related to the reallocation of `string` and `vector`. Portability issues of the associative containers are also investigated. I have examined what kind of problems can be occured due to the lack of virtual destructors.

In the dissertation I review the iterator-related problems: the invalidation and conversion connected ones. I show the problems connected to the confusion of pointers and iterators as well.

I overview the problems related to the functors and allocators. Besides, I show some complex error diagnostics. I also investigate header-related issue that results in portability problems.

4 Objectives

My objective was to help the programmers by detecting the erroneous STL usage expansively and assist the detection with formal and software tools, as well. However, an experimental tool, STLlint [4] was available for a long time, but it did not live up to expectations, so its support was cancelled. STLlint is based on a modified compiler. It works according to the compile-time information solely. In contrast, my solutions are based on extension and modification of the library with usage of standard compliant compilers. I also prefer the detection of possible problems at compilation time and generate warnings with the help of C++ templates, but some of my approaches work at runtime. My objectives are defined in the form of the hereinafter priorities:

1. Detecting the potential errors of STL misusages *at compile time in a non-intrusive way.*

2. Detecting the potential errors of STL misusages *at compile time with the modification of the library.*

3. Detecting the potential errors of STL misusages *at run-time in a non-intrusive way and keeping the asymptotic limit of the STL specification* (with minimal overhead).

4. Detecting the potential errors of STL misusages *at run-time with the modification of the library and keeping the asymptotic limit of the STL specification*(with minimal overhead).

5. Detecting the potential errors of STL misusages *at run-time in a non-intrusive way and breaking the asymptotic limit of the STL specification.*

6. Detecting the potential errors of STL misusages *at run-time with the modification of the library and breaking the asymptotic limit of the STL specification.*

5 Formal approach of the STL

The specification of the C++ programming language includes the specification of STL. However, STL's specification is informal [14]. This may be ambiguous and makes the correctness examinations hard.

I have shown two kinds of formal tools that can be used for the specification of STL. I have worked out a technique that uses *pre- and postconditions* for the specification of STL. This technique is based on the well-known Hoare's method. It is utilizable for STL-based programs, libraries and STL implementations, as well. The other one specification is written in *LaCert* language and uses the tools of temporal logic. The main objective of this approach is generation of STL-based code by the LaCert compiler at the critical points. I have taken part in the completion of the specification, but this work belongs to Gergely Dévai.

These specifications can be used for verification of STL-based programs or libraries and detection of incidental errors. Verification of STL implementation can be done with these tools, as well. STL-based code can be generated that formally proven with the integration of LaCert specifications.

1st Thesis. I have developed a tool system that is suitable for formal specification of generic programs. I have shown its suitability by the formal

specification of the STL basic components. The definition of library can be more exact with this method and even in minor cases it can be used in correctness examinations.

Some important publications that belong to this thesis: [2, 3, 6, 14, 16].

6 Compile-time approaches

Compilers cannot emit warnings when the STL is in-use with potential semantical errors [21]. An important advantage of compile-time solutions that the runtime of compiled code is unchanged, thus the effectiveness of the STL remains and the implementation still keeps the specification.

I have developed tools that detect the incorrect usage of STL at compilation time. The compiler emits compilation warning if one of the constructs likely behaves erroneously during execution. I have developed a method that is able to generate "custom" warnings. I do not change the compiler itself, but I modify the source code of the library.

Believe-me marks are *annotations* that do not result in runtime activity, but disable the warnings that I generate. I have developed believe-me marks to the warnings. Programmers can figure out the correctness of the code with these annotations.

2nd Thesis. I have developed methods that are able to detect some particular cases of STL misuses at compilation time. The method is based on the modification of the library, so my approaches can be used with every standard-compliant compilers. The methods belong to the erroneous instantiations (2.1), certain algorithms (2.2), adaptable functors (2.3), allocators (2.4), reverse iterators (2.5) and lazy instantiation (2.6).

Erroneous instantiations	[10]
Algorithms	[15]
Adaptable functors	[8]
Allocators	[13]
Reverse iterators	[13]
Lazy instantiation	[9]

Table 1: Some important publications related to the thesis

7 Run-time approaches

Unfortunately, at compilation time not all information is available to make sure if a potential error actually results in a runtime problem. At runtime we have more pieces of information to answer these questions, give signals for the users or correct the behavior of the system. In these cases the verifications are evaluated at runtime, thus the execution can slow significantly. I have developed different methods, components that increase the safety of the libary usage and work at runtime.

3rd Thesis. I have developed methods that are able to detect or avoid some particular cases of STL misuses at runtime. The methods belong to the iterator invalidation (3.1), copy-safe iterators (3.2), erasable iterators (3.3), precondition of certain algorithms (3.4) and the usage of functors (3.5)

Invalid iterators	[12, 18, 19]
Copy-safe iterators	[11, 12]
Erasable iterators	[11, 12]
Precondition of algorithms	[12, 18, 19]
Functors	[8, 9]

Table 2: Some important publications related to the thesis

References

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

[2] 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.

[3] Dévai, G., Pataki, N.: *A tool for formally specifying the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **31** (2009), pp. 147–166.

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

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

[6] Pataki, N.: *A C++ Standard Template Library helyessgvizsglata* (TDK Dolgozat), Orszgos Tudomnyos Dikkri Konferencia (2005).

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

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

[9] 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.

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

[11] Pataki, N.: *Advanced Safe Iterators for the C++ Standard Template Library*, in Proc. of the Eleventh International Conference on Informatics, Informatics 2011, pp. 86-89.

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

[13] 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.

[14] Pataki, N., Dévai, G.: *A Comparative Study of C++ Standard Template Library's Formal Specification*, in Conference of PhD Students in Computer Science, CSCS 2008, Volume of extended abstracts, 2008, p. 48.

[15] 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 2011), pp. 919–922.

[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., Szűgyi, Z.: *C++ Exam Methodology*, Annales Mathematicae et Informaticae **37** (2010), pp. 211–223.

[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 Theoretical Computer Science (ENTCS) Vol. **264(5)** (2011), pp. 71–83.

[20] Stroustrup, B., The C++ Programming Language (Special Edition), *Addison-Wesley* (2000).

[21] Van Wyk, E., Borin, D., Huntington, P.: *Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions*, in Proc. of the Second International Workshop on Library-Centric Software Design (LCSD '06), pp. 35–44.