Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Software for the Refined Version of the Quality Assurance Tool. D3.3

Due date of deliverable: 24

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP3

Responsible institution: PRQA
Editor and editor's address: Evgueni Kolossov, PRQA

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary

This deliverable reports on the Task 3.4 Quality Assurance Analysis with some elements from the tasks T2.3, T3.2, T3.3, and T3.5. In this tasks we extended the PRL QA-C++ and QA-Verify Quality Assurance tool in order to develop new analyses that are capable of automatically analysing source code and verify its compliance to code and data standards that have been developed in T5.3. In particular we introduce full support for C++'11, and C++'14 together with some initial implementation of support for C++'17 language standards. Together with implementation of support for new versions of C++ language we have implemented particular checks into compliance module which is checking for the compliance to "High Integrity C++ for Parallel and Concurrent Programming" coding standard. QA-Verify have been updated to support new compliance module and new functionality in QA-C++. Refined QA-C++ version currently run on regular base for all RePhrase use cases and producing regular output available for all members of consortium via QA-Verify updated tool. Preview version of new Compliance module running in test mode to receive output and amend it based on that.

Contents

Chapter 1

Introduction

Most of applications using parallelisation in C++ are written in C++'14 (sometimes even in C++'17) and this means that PRL tools need to have full support for this version of the C++ language. For this reason, we have been concentrating on the implementation of full support for C++'14. During this task we have managed to achieve full coverage for C++'14, some initial implementation for C++'17 functionality, and some outstanding functionality for C++'11. Currently the preview version of QA-C++ 4.2 has all this functionality. The release for customers is planned for end of July 2017 but the preview version has already been run on RePhrase use cases projects on a regular basis. The second target for this period has been the creation of an initial version of the new compliance module for "High Integrity C++ for Parallel and Concurrent Programming" standard developed during this project (final version will be ready for deliverable 5.5 by end of June 2017). Initial version of this compliance module currently run in test mode for RePhrase use cases projects.

Chapter 2

Support for latest C++ versions

2.1 Support for C++ '14 variable templates

reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3651.pdf. C++ has no notation for parameterized constants as direct as for functions or classes. For instance, we would like to represent the mathematical constant pi with precision dictated by a floating point datatype

```
template<typename T>
    constexpr T pi = T(3.1415926535897932385);
```

and use it in generic functions, e.g. to compute the area of a circle with a given radius:

```
template<typename T>
    T area_of_circle_with_radius(T r) {
        return pi<T> * r * r;
}
```

This is new functionality implemented.

2.2 Support for C++ '11 constant expressions (for static member and non-member functions declared as constexpr)

Reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf. This is related to the notion of constant expressions to include constant expression functions and user-defined literals. In addition, some floating point constant expressions are allowed. The goal is to improve support for generic programming, systems programming, and library building, and to increase C99 compatibility. The implementation is targeting to remove long-standing em-

barrassments from some Standard Library components (notably <limits>). The standard

```
numeric_limits
```

functionality of <limits.h>, but fails to deliver constant expressions. For example,the expression

```
numeric_limits<int>::max()
```

to the macro

```
INT_MAX
```

The main approach to this implementation suggests to allow explicitly identified simple functions to be used as part of constant expressions.

This is new functionality implemented.

2.3 Support for C++ '14 contextual implicit conversions

Reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3323.pdf. The context in which a C++ expression appears often influences how the expression is evaluated, and therefore may impose requirements on the expression to ensure such evaluation is possible. For example, it is well-understood that an expression used in an if, while, or similar context must be convertible to bool so that the expression can be converted to bool during its evaluation. This conversion is termed contextual, and is set forth in [conv]/3. In four cases, the FDIS (N3290) uses different language to specify an analogous context dependent conversion. In those four contexts, when an operand is of class type, that type must have a single non-explicit conversion function to a suitable (context-specific) type.

This is new functionality implemented.

2.4 Support for C++ '14 non static data member initializers for aggregate initialization

Reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3653.html. Implementation is related to the situation when an aggregate is an array or a class with no user-provided constructors, no private or protected non-static data members, no base classes, and no virtual functions. If there are fewer initializer-clauses in the list than there are members in the aggregate, then each member not explicitly initialized shall be initialized from its brace-or-equal-initializer or, if there is no brace-or-equal-initializer, from an empty initializer list.

This is new functionality implemented.

2.5 Support for C++ '14 sized deallocation functions

Reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3778.html. There is a problem with C++11, where programmers may define a static member function operator delete that takes a size parameter indicating the size of the object to be deleted. The equivalent global operator delete is not available. This omission has unfortunate performance consequences. Modern memory allocators often allocate in size categories, and, for space efficiency reasons, do not store the size of the object near the object. Deallocation then requires searching for the size category store that contains the object. This search can be expensive, particularly as the search data structures are often not in memory caches. The solution to this problem is to permit implementations and programmers to define sized versions of the global operator delete. The compiler shall call the sized version in preference to the unsized version when the sized version is available.

This is new functionality implemented.

2.6 Support for "static_assert" declarations according to C++ '11 and C++ '17

Reference: http://en.cppreference.com/w/cpp/language/static_assert. Static assertion performs compile-time assertion checking. A static assert declaration may appear at namespace and block scope (as a block declaration) and inside a class body (as a member declaration) If bool_constexpr returns true, this declaration has no effect. Otherwise a compile-time error is issued, and the text of message, if any, is included in the diagnostic message. Since message has to be a string literal, it cannot contain dynamic information or even a constant expression that is not a string literal itself. In particular, it cannot contain the name of the template type argument. Message can be omitted since C++17.

This is new functionality implemented.

2.7 Support for C++ '11 decltype of call expression with incomplete return type

Reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3276.pdf. This implementation related to the US22/DE9, which mistakenly attributed a type-completeness requirement to any expression argument of decltype, but which was nevertheless trying to call attention to a real problem: that when the expression of a decltype-specifier is a function call expression, the longstanding requirement for a call-expression's type to be complete (from 5.2.2 [expr.call]) is both unnecessary and harmful. In that limited context, this old requirement causes unexpected and catastrophic problems. The committee decided

in Rapperswil that this issue was NAD. The issues caused by that requirement are real and very serious, possibly leading to poor adoption of decltype and result_of.

This is new functionality implemented.

2.8 for C++ '11 unrestricted unions

Reference: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2544.pdf. When confronted with functionality that can be misused, a language design must often make a choice between prohibiting it in the interests of safety, and allowing it in the interests of flexibility and power. Generally speaking, C++ takes the latter approach. However, in the case of unions C++ diverges from this philosophy and places severe limitations on members in the interest of safety. This limitation seems unnecessary, and without it unions can become very useful tools for solving certain problems. Many important problem domains require either large numbers of objects or limited memory resources. In these situations conserving space is very important, and a union is often a perfect way to do that. In fact, a common use case is the situation where a union never changes its active member during its lifetime. It can be constructed, copied, and destructed as if it were a struct containing only one member. A typical application of this would be to create a heterogeneous collection of unrelated types which are not dynamically allocated (perhaps they are in-place constructed in a map, or members of an array). Unfortunately most objects cannot be members of unions. Good object programming practice results in constructors for even simple, lightweight, transparent types. For example a point type for geographic work needs a good set of constructors, but such well designed types cannot be put into unions. There are no any reason to prohibit this. Implemented solution is to remove all of the restrictions on the types of members of unions, with the exception of reference types. Any non-reference type may be a union member. New standard have removed the restriction on static members (for non-anonymous unions) as well since there appears to be no reason for this restriction, and static members are as useful for unions as for other class types. New standard have also changed the way implicitly declared special member functions of unions are generated in the following way: if a non-trivial special member function is defined for any member of a union, or a member of an anonymous union inside a class, that special member function will be implicitly deleted for the union or class. This prevents the compiler from trying to write code that it cannot know how to write, and forces the programmer to write that code if it is needed. The fact that the compiler cannot write such a function is no reason not to let the programmer do so. The current standard specifies that unions have all their members automatically defaultinitialized (as in a struct) through the automatic initialization of members not named in the mem-initializer-list. This is technically wrong since only one member can be active at once, but does not cause a problem in practice since the constructors of all members must be trivial. Since we are allowing non-trivial members, we need to fix this by suppressing the au-

tomatic initialization of members for unions. (The current standard also specifies memberwise copying of trivial unions, but this will be the subject of a separate core issue.) New standard also suppress the automatic destruction of union members. An implication of this is that because members are not automatically initialized, if the programmer fails to list the desired active member in the mem-initializer-list, it will not be constructed. This may or may not be an error since the body of the constructor may take care of the member construction, but the compiler cannot help here. A subtle implication of allowing non-trivial members is that programmers can no longer always use copy assignment to change the active field. Since it is in not in general valid to assign to an unconstructed object, changing fields may require destruction and construction. For example, consider a union u of type U containing a member m of type M and a member n of type N. If M has a non-trivial destructor and N has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member must be switched from m to n using the destructor and placement new operator. So the N assignment operator might look like this:

```
U& U::operator=(const N& new_n)
{
 m.~M();
 new (&n) N(new_n);
 return *this;
}
```

Chapter 3

Fixes for Problems found

During initial implementation of support for the latest versions of C++ language and RePhrase standard a few problems have been identified and fixed during this period:

3.1 False positive message 4212 for member function with pseudo destructor call expression

PROBLEM DESCRIPTION: QAC++ is not detecting the use of 'this' where the body of a member contains pseudo destructor calls.

SAMPLE:

```
<code>
  template < typename T >
  class svector {
  public :
    inline void pop_back ( ) { first -> ~ T ( ) ; }
       (cont.)   // expect: !4212
  private :
    T * first ;
    int len ;
  } ;

  template class svector<int>;

  class B {
    typedef int TYPE;
    TYPE * p;
    void bar () {   // expect: !4212
      p->~TYPE();
    }
```

```
  };
</code>
```

CLOSE CRITERIA: QAC++ shall not generate 4212 for a member function that does nothing but contain a pseudo destructor call.

Problem successfully resolved.

3.2 False negative messages for side-effects to locals with static storage duration as well as improved modelling for virtual calls including consideration of final members and classes

ENHANCEMENT DESCRIPTION: QAC++ is not warning when a side-effect is used as part of

`std::conditional_variable::wait.`

Additionally, there are some other cases which do not issue the correct message, for example modification of a local object with static duration should be seen as a side-effect.

SAMPLE:

```
<code>
  #include <mutex>
  #include <condition_variable>
  #include <cstdint>

  std::mutex mut;
  std::condition_variable cv;

  int32_t i;

  bool sideEffects()
  {
    ++i;
    return (i > 10);
  }

  void threadX()
  {
    i = 0;
    std::unique_lock<std::mutex> guard(mut);
    cv.wait(guard, sideEffects);  // expect: XXXX
  }
```

```
</code>

<code>
  bool foo ()
  {
    static int i;
    ++i;
    return i % 2;
  }

  void bar ()
  {
    1 && foo();  // expect: 3227,3230,!3231
  }
</code>
```

CLOSE CRITERIA: QAC++ shall treat arguments to

`'conditional_variable::wait'`

Problem successfully resolved.

3.3 False positive message 4212 for member function with pseudo destructor call expression

PROBLEM DESCRIPTION: QAC++ is not detecting the use of 'this' where the body of a member contains pseudo destructor calls.

SAMPLE:

```
<code>
  template < typename T >
  class svector {
  public :
    inline void pop_back ( ) { first -> ~ T ( ) ; }
       (cont.)   // expect: !4212
  private :
    T * first ;
    int len ;
  } ;

  template class svector<int>;

  class B {
    typedef int TYPE;
```

```
    TYPE * p;
    void bar () {   // expect: !4212
      p->~TYPE();
    }
  };
</code>
```

Problem successfully resolved.

Chapter 4

Conclusion

Support for the latest versions of C++ language is extremely important for static analyser to identify possible hygienic, maintenability, security, performance and other violations of the language and standards. During this period PRL have achieved full coverage for C++'11 and C++'14 with some implementation of support for C++'17 versions of the language. This implementation have made PRL tool to be able to analyse comprehensive parallelisation code and provide diagnostics for violation of new standard.