Project no. 644235

REPHRASE

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

Software for the final version of the QA tool
D3.4

Due date of deliverable: 31 December 2017

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP3

Responsible institution: Programming Research Ltd.
Editor and editor's address: Evgueni Kolossov, Programming Research Ltd.

Version 1.0

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 Tasks 3.4 Quality Assurance Analysis with some elements from the tasks T2.3, T3.2, T3.3, and T3.5 Detection of Extra Functional Property Violations. 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 and D5.5. 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.

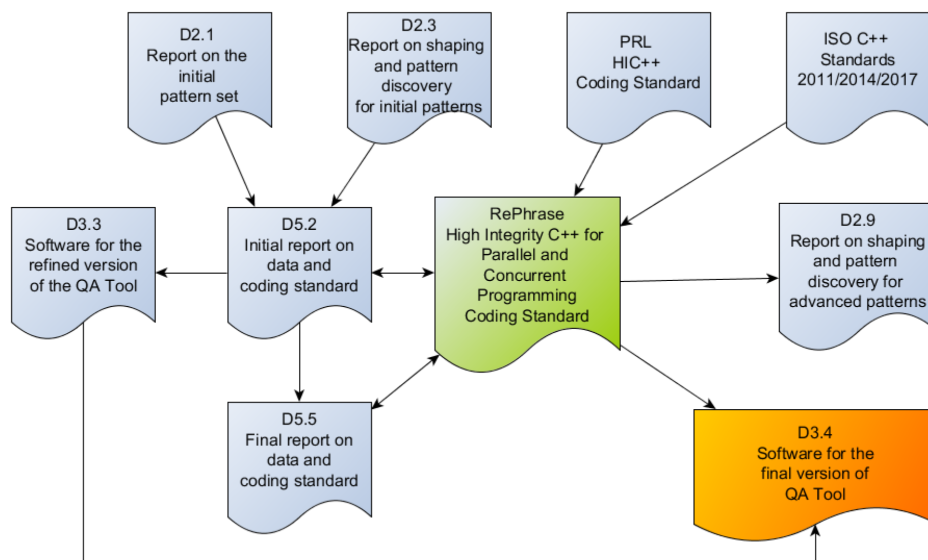

Figure 1: Dependencies of D3.4

Contents

Executive Summary 1

1 Introduction **3**

2 Improved support for parsing of the latest version of C++ **4**
2.1 Lookup of friend class names in class templates 4
2.2 Delay substitution of unexpanded parameter packs into alias templates 5
2.3 SFINAE for list initialisation 5
2.4 Allow overloading of template functions on the return type 6

3 Coding Standard Enforcement **8**
3.1 Rule 3.1 Do not use platform specific multi-threading facilities . . 8
3.2 Rule 3.2 Use `std::async` instead of `std::thread` for task-based parallelism 9
3.3 Rule 3.3 Always explicitly specify a launch policy for std::async . 9
3.4 Rule 3.7 Shared global data shall be provided through static local objects 10
3.5 Rule 3.8 Use std::call_once to ensure a function is called exactly once 11
3.6 Rule 3.9 Use a noexcept lambda as the argument to a thread constructor 11
3.7 Rule 3.10 The lifetime of data passed to the thread constructor must exceed the lifetime of the thread 12
3.8 Rule 3.11 Within the scope of a lock, ensure that no static path results in a lock of the same mutex 13
3.9 Rule 3.12 Ensure that order of nesting of locks in a project forms a DAG 14
3.10 Rule 3.13 Do not use std::recursive_mutex 15
3.11 Rule 3.14 Objects of type std::mutex shall not be accessed directly. 16
3.12 Rule 3.15 Objects of type std::mutex shall not have dynamic storage duration. 16
3.13 Rule 3.16 Do not use relaxed atomics 17

3.14 Rule 3.17 Do not use std::condition_variable_any on objects of type std::mutex . . . 18
3.15 Rule 4.1 Use higher-level standard facilities to implement parallelism 19
3.16 Rule 4.2 Functor used with a parallel algorithm shall be pure . . . 19
3.17 Rule 4.4 This hygienic sequential algorithm can be replaced with a parallel version . . . 20
3.18 Rule 4.5 Functor used with a parallel algorithm shall always return 20
3.19 Rule 4.6 Functors used with parallel algorithms shall be noexcept 21
3.20 Rule 4.9 The `Function` argument used with an algorithm shall not use a non-const iterator or reference to the container being iterated over. . . . 22

4 Fixes for Problems found **24**
4.1 Messages 4113 and 4114 are no longer generated for a declaration from a macro expansion. . . . 24
4.2 False positive for a call to a non-static member template function of the enclosing class in a lambda expression if it has explicitly specified template arguments. . . . 24

5 Conclusion **26**

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, and some initial implementation for C++'17 functionality. Currently the release version of QA-C++ 4.2 has all this functionality. Product released for customers in November 2017. Currently the most of the work concentrated on the next release - 4.2.1 with implementation of RePhrase coding standard which is planned to release for customers in April 2018 but the preview version has already been run on RePhrase use cases projects on a regular basis producing valuable diagnostics of non-compliance to the standard. .

Chapter 2

Improved support for parsing of the latest version of C++

In addition to significant features being added to the C++ language between versions of the standard, like C++'11 and C++'14, some of the language features get clarified or modified as a result of feedback from implementation and usage experience. These changes and clarafications need to be considered in QA-C++ as well to be able to handle any edge-cases encountered during parsing of modern C++ code.

2.1 Lookup of friend class names in class templates

Enhancement Description:

Changed lookup of friend class names in class templates to occur at template definition time instead of template instantiation time to avoid inadvertently looking up in dependent base classes.

Example:

```
template<typename T> class A
{ };

template<typename T>
struct F
{
  template <typename T1, typename T2>
  class A
  { };
};

template<typename T>
struct B
```

```
  : F <T>
{
  friend class A <T>; // lookup finds ::A
};

void foo()
{
  B<int> b;
}
```

Resolution:

Handling of name lookup for friend classes of class templates has been updated in the QA-C++ parser.

2.2 Delay substitution of unexpanded parameter packs into alias templates

Enhancement Description:

Substitution of alias templates needs to be delayed until any template parameter packs in template arguments are fully expanded.

Example:

```
template<typename T> using Int = int;

template<typename ...Ts> struct S
{
  // OK: not attempting to substitute S<int ...>
  typedef S<Int<Ts> ...> other;
};

template struct S<char, short, int>;
```

Resolution:

Handling of substitution of alias templates with unexpanded template parameter packs has been updated in the QA-C++ parser.

2.3 SFINAE for list initialisation

In general, QA-C++ tries to recover from non critical parsing errors to continue analysis for as much code as possible. However, with C++'11 list initialisation can now also be used in SFNAE (Substitution Failure is Not An Error) contexts where recovery isn't desirable.

Example:

```
struct A
{
  int i, j;
};

template<typename T, int ... I>
decltype(T{ I ... }) foo(int);

template<typename T, int ... I>
int foo(int);

void bar()
{
  // calls 'int foo(int);'
  int i = foo<A, 1, 2, 3> (1);
}
```

Resolution:

Handling of list initialization errors in SFNAE contexts has been updated in the QA-C++ parser.

2.4 Allow overloading of template functions on the return type

Enhancement Description:

Unlike non-template functions, template functions can also be overloaded on the return type, even if the return type is non-dependent.

Example:

```
template<typename T>
void foo(int)
{ }

template<typename T>
int foo(int) // OK: valid overload
{
  return 0;
}

void bar()
{
  int (*fn) (int) = foo<int>;
}
```

Resolution:

Handling of overloading of template functions has been updated in the QA-C++ parser to allow for overloading on the return type.

Chapter 3

Coding Standard Enforcement

3.1 Rule 3.1 Do not use platform specific multi-threading facilities

Enhancement Description:

Rather than using platform-specific facilities, the C++ standard library should be used as it is platform independent.

Example:

```
// Non-Compliant
#include <pthread.h>
void* thread1(void*);
void f1()
{
  pthread_t t1;
  pthread_create(&t1, nullptr, thread1, 0);
  // ...
}

// Compliant
#include <thread>
void thread2();
void f2()
{
  std::thread t1(thread2);
  // ...
}
```

Resolution:

New analysis has been implemented in the Compliance Module by diagnosing calls to platform-specific functions.

3.2 Rule 3.2 Use `std::async` instead of `std::thread` for task-based parallelism

Enhancement Description:

`std::thread` is a low-level facility whose destructor will call `std::terminate` if the thread owned by the class is still joinable. It should therefore only be used for detached threads or in conjunction with an application specific cancellation mechanism that signals pending termination to the thread before joining it.

For simple task-based parallelism `std::async` with a launch policy of `std::launch::async` provides a better abstraction.

Example:

```
#include <thread>
#include <future>

void f(int);
int main()
{
  int i = 0;

  // Non-Compliant: Potentially calls 'std::terminate'
  std::thread t(f, i);

  // Compliant: Will wait for the task to complete
  auto r = std::async (std::launch::async, f, i);
}
```

Resolution:

New analysis has been implemented in the Compliance Module by diagnosing calls to the `std::thread` constructor.

3.3 Rule 3.3 Always explicitly specify a launch policy for std::async

Enhancement Description:

The default launch policy for `std::async` leaves it to the implementation to choose between `async` and `deferred` as the launch policy. This could result in code that expects to be executed asynchronously not being executed asynchronously.

Example:

```
#include <future>
```

```
void f(int);
int main()
{
  int i = 0;

  // Non Compliant: Uses default launch policy
  auto f1 = std::async (f, i);

  // Compliant: Uses async launch policy
  auto f2 = std::async (std::launch::async, f, i);
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser by checking that only overloads of `std::async` taking `std::launch` as the first parameter are used.

3.4 Rule 3.7 Shared global data shall be provided through static local objects

Enhancement Description:

The ISO C++ Standard guarantees that initialization of local static data is correctly synchronized between threads. Use of local statics avoids the need for constructs such as `call_once` or the more complex Double Checked Locking Pattern.

Example:

```
// Non-Compliant - complex initialisation required
int * global_instance;
int & getInstance1 () {
  return *global_instance;
}

// Compliant
int * init () {
  return new int (0);
}

int & getInstance2 () {
  static int * instance (init ());
  return *instance;
}
```

Resolution:

New analysis has been implemented in RCMA (Cross-Module Analysis) module.

3.5 Rule 3.8 Use std::call_once to ensure a function is called exactly once

Enhancement Description:

The standard library provides the `std::call_once` that can be used to guarantee that a call is made at most once.

Example:

```
#include <mutex>

namespace
{
  std::once_flag startup_called;
}

void startup (const char *)
{
  // ...
}

void thread_start()
{
  // Compliant: Using 'call_once'
  std::call_once (startup_called, startup, "Hello␣
     (cont.)World");
  // ...
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser by detecting any use of the double checked locking pattern in the source code.

3.6 Rule 3.9 Use a noexcept lambda as the argument to a thread constructor

Enhancement Description:

Consideration must be made for the lifetime of parameters passed to threads. Using a lambda can be used as the entry point of the thread allows explicit control over which variables are captured for use in the thread.

Similarly, for reference arguments, the lambda syntax provides a simple and consistent method for capturing them, without the need to wrap them in std::ref.

Example:

```
#include <thread>

void worker(int);
void foo()
{
  int i;

  // Non-Compliant
  std::thread t1 ( worker, i );

  // Compliant
  std::thread t2 ( [i]() noexcept {
    try
    {
      worker(i);
    }
    catch (...)
    {
    }
  });

  t1.join ();
  t2.join ();
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser by checking that the first parameter to `std::threads` constructor (that is not a move constructor) is a lambda expression with a `noexcept` specifier.

3.7 Rule 3.10 The lifetime of data passed to the thread constructor must exceed the lifetime of the thread

Enhancement Description:

Any objects being passed to a new thread by reference need to outlive the thread.

If the thread is being detached or the thread object is being returned from the function (or added to a container with a longer lifetime), it is likely that the thread outlives any objects in the local scope.

Example:

```
#include <thread>

void worker(int *);
void f1 () {
  int i;
  std::thread t1 ( [&i](){ worker(&i); } );
  // Non-Compliant: Lifetime of thread possibly
     (cont.)exceeds that of 'i'
  t1.detach ();
}
```

Resolution:

New analysis has been implemented in the parser by checking that a lambda expression passed as the first argument to `std::threads` constructor does not capture any objects by reference if the thread is detached or moved to an object with greater lifetime.

3.8 Rule 3.11 Within the scope of a lock, ensure that no static path results in a lock of the same mutex

It is undefined behavior if a thread tries to lock a std::mutex it already owns, this should therefore be avoided.

Example:

```
#include <mutex>

std::mutex mut;
int i;

void f2(int j);

void f1(int j) {
  std::lock_guard<std::mutex> hold(mut);
  if (j) {
    f2(j);
  }
  ++i;
}

void f2(int j) {
  if (! j)
  {
```

```
    // Non-Compliant: "Static Path" Exists to here
       (cont.)from f1
    std::lock_guard<std::mutex> hold(mut);
    ++i;
  }
}
```

Resolution:

New analysis has been implemented in the RCMA (Cross-Module Analysis) module.

3.9 Rule 3.12 Ensure that order of nesting of locks in a project forms a DAG

Enhancement Description:

Mutex locks are a common causes of deadlocks. Multiple threads trying to acquire the same lock but in a different order may end up blocking each other.

When each lock operation is treated as a vertex, two consecutive vertices with no intervening lock operation in the source code are considered to be connected by a directed edge. The resulting graph should have no cycles, i.e. it should be a Directed Acyclic Graph (DAG).

Example:

```
#include <mutex>

// Non-Compliant:  Nesting of locks does not form a
   (cont.)DAG:
//                 mut1->mut2 and then mut2->mut1
class A
{
public:
  void f1() {
    std::lock_guard<std::mutex> lock1(mut1);
    std::lock_guard<std::mutex> lock2(mut2);
    ++i;
  }

  void f2() {
    std::lock_guard<std::mutex> lock2(mut2);
    std::lock_guard<std::mutex> lock1(mut1);
    ++i;
  }
```

```
private:
  std::mutex mut1;
  std::mutex mut2;
  int i;
};
```

Resolution:

New analysis has been implemented in the RCMA (Cross-Module Analysis) module.

3.10 Rule 3.13 Do not use std::recursive_mutex

Use of std::recursive_mutex is indicative of bad design: Some functionality is expecting the state to be consistent which may not be a correct assumption since the mutex protecting a resource is already locked.

Example:

```
// Non-Compliant: Using recursive_mutex
#include <mutex>

class DataWrapper
{
public:
  int incrementAndReturnData() {
    std::lock_guard<std::recursive_mutex> guard(mut);
    incrementData();
    return data;
  }

  void incrementData() {
    std::lock_guard<std::recursive_mutex> guard(mut);
    ++data;
  }

  // ...
private:
  mutable std::recursive_mutex mut;
  int data;
};
```

Resolution:

New analysis has been implemented in the Compliance Module by diagnosing calls to `std::recursive_mutex`s constructor.

3.11 Rule 3.14 Objects of type std::mutex shall not be accessed directly.

Enhancement Description:

As for other resource types, it is important that locked mutexes are released at the end of the critical section. The simplest approach to this is using RAII and `std::lock_guard`.

Example:

```
#include <mutex>

std::mutex mut;

void doSomethingCritical1() {
  // Non-Compliant
  mut.lock ();
  // critical code here
  mut.unlock ();
}

void doSomethingCritical2() {
  std::lock_guard<std::mutex> guard(mut);  //
     (cont.)Compliant
  // critical code here
  // lock automatically released
}
```

Resolution:

New analysis has been implemented in the Compliance Module by diagnosing calls to member functions `lock`, `try_lock` and `unlock` of `std::mutex`.

3.12 Rule 3.15 Objects of type std::mutex shall not have dynamic storage duration.

Enhancement Description:

It is undefined behavior to destroy a locked mutex. Declaring objects with type `std::mutex` in global scope or as a function local static and accessing them exclusively through the use of `std::lock_guard` will avoid the danger of destroying a mutex that is currently locked.

Example:

```
#include <mutex>
```

```
std::mutex * mut = new std::mutex; // Non Compliant

void doSomethingCritical()
{
  std::lock_guard<std::mutex> guard(*mut);
  delete mut;  // Undefined behaviour
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser by checking for new expressions of type `std::mutex`.

3.13 Rule 3.16 Do not use relaxed atomics

Enhancement Description:

Using non-sequentially consistent memory ordering for atomics allows the CPU to reorder memory operations resulting in a lack of total ordering of events across threads. This makes it extremely difficult to reason about the correctness of the code.

Example:

```
#include <atomic>

template<typename T>
class CountingConsumer
{
public:
  explicit CountingConsumer(T *ptr, int cnt)
    : m_ptr(ptr), m_cnt(cnt) { }

  void consume (int data) {
    m_ptr->consume (data);

    // Non-Compliant
    if (m_cnt.fetch_sub (1, std::memory_order_release)
       (cont.) == 1) {
      delete m_ptr;
    }
  }

  T * m_ptr;
  std::atomic<int> m_cnt;
};
```

Resolution:

New analysis has been implemented in the QA-C++ parser by checking for calls to atomic operations with a memory order argument that isn't `std::memory_order_seq_cst`.

3.14 Rule 3.17 Do not use std::condition_variable_any on objects of type std::mutex

Enhancement Description:

When using `std::condition_variable_any`, there is potential for additional costs in terms of size, performance or operating system resources, because it is more general than `std::condition_variable`.

`std::condition_variable` works with `std::unique_lock`, while `std::condition_variable_any` can operate on any objects that have lock and unlock member functions.

Example:

```
#include <mutex>
#include <condition_variable>
#include <vector>

std::mutex mut;
std::condition_variable_any cv;
std::vector<int> container;

void producerThread()
{
  int i = 0;
  std::lock_guard<std::mutex> guard(mut);

  // critical section
  container.push_back(i);

  cv.notify_one();
}

void consumerThread()
{
  std::unique_lock<std::mutex> guard(mut);

  // Non-Compliant: conditional_variable_any used with
  //                std::mutex based lock 'guard'
  cv.wait(guard, [](){ return !container.empty(); } );
```

```
  // critical section
  container.pop_back();
}
```

Resolution:

New analysis has been implemented in the Compliance Module to point to calls of `std::condition_variable_any` member functions with a parameter type of `std::unique_lock<std::mutex>`.

3.15 Rule 4.1 Use higher-level standard facilities to implement parallelism

Enhancement Description:

Low-level threading facilities like std::thread should not be used to implement parallelism as it can be difficult to achieve both correctness and performance. Instead, higher-level abstractions based on well-known parallel patterns should be used to implement parallelism.

Parallel versions of most STL algorithms have been included in C++'17 (and the Parallism TS) which should be preferred in comparison to any approach based on low-level threading facilities.

Resolution:

New analysis has been implemented in the Compliance Module to point to the use of low-level thread functions.

3.16 Rule 4.2 Functor used with a parallel algorithm shall be pure

Enhancement Description:

A function is non-pure if it:

- reads or writes to an object other than:
  - a non-volatile automatic variable, or
  - a function parameter, or
  - an object allocated within the body of the function.
- calls a non-pure function.

Example:

```
int * f2(int i, int j)
{
  // Compliant
```

```
  int * k = new int (i + j);
  return k;
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser to check that any functors passed to parallel algorithms are pure.

3.17 Rule 4.4 This hygienic sequential algorithm can be replaced with a parallel version

Enhancement Description:

If a functor is hygienic then it can be used with either sequential or parallel algorithms. This is an example of an algorithm that can be switched to the parallel kind.

Example:

```
#include <algorithm>
#include <execution>

bool foo()
{
  int arr[] = { 1, 2, 3, 4, 5 };

  // Compliant
  bool b = std::all_of(std::execution::seq
    , std::cbegin(arr)
    , std::cend(arr)
    , [] (int i) noexcept { return i >= 0; });

  return b;
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser to point to the use of sequential algorithms where the functor passed in is pure.

3.18 Rule 4.5 Functor used with a parallel algorithm shall always return

Enhancement Description:

A function will not be considered as pure when:

- The body contains loops which can be statically determined as infinite, or
- The function can be statically determined as directly or indirectly recursive
- The function calls a function that causes the program to exit immediately, for example:
  - std::exit
  - std::abort
  - std::terminate
- The function calls std::longjmp

Example:

```
#include <cstdlib>

class DoSomething1 {
  // Non-Compliant
  void operator()(int * i) {
    if (! i) {
      std::exit (1);
    }
  }
};
class DoSomething2 {
  // Non-Compliant
  void operator()(int * i) {
    while (true) ;
  }
};
```

Resolution:
New analysis has been implemented in the Compliance Module to point to infinite loops that have been identified and calls to `exit`, `abort` and `std::terminate`.

3.19 Rule 4.6 Functors used with parallel algorithms shall be noexcept

Enhancement Description:
An exception thrown from an element access function used with a parallel algorithm will result in `std::terminate` being called. Addition of `noexcept` exception specification documents this explicitly. The intention is that catch handlers, alternatively, manual or automated analysis will be used to ensure exceptions do not propagate out of the functor.

Example:

```
#include <algorithm>
#include <execution>
#include <vector>

void f(std::vector<int> & v)
{
  try
  {
    // Non Compliant
    std::for_each (std::execution::seq
      , v.begin ()
      , v.end ()
      , [](int) { throw 0; } );
  }
  catch (int & e)
  {
  }
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser to check that a functor passed to a parallel algorithm has been declared `noexcept`.

3.20 Rule 4.9 The `Function` argument used with an algorithm shall not use a non-const iterator or reference to the container being iterated over.

Enhancement Description:

It is undefined behavior if the `unary_op` or `binary_op` modifies any of the iterators in a container while iterating over the same container.

Example:

```
#include <vector>
#include <algorithm>

void foo (std::vector<int> & v) {
  // Non Compliant, non const reference
  // passed to the container
  std::for_each (v.begin ()
    , v.end ()
    , [&v](int i) { v.erase(v.begin ()); });
```

```
  // Compliant, const reference.
  auto const & cv (v);
  std::none_of (v.begin ()
    , v.end ()
    , [&cv](int i) { return i >= cv.size (); });
}
```

Resolution:

New analysis has been implemented in the QA-C++ parser to check that the functor (if it is a lambda expression) passed to an algorithm does not modify the container the algorithm is iterating over.

Chapter 4

Fixes for Problems found

4.1 Messages 4113 and 4114 are no longer generated for a declaration from a macro expansion.

Problem Description:

When code originates from a macro expansion, it's not always possible to address problems found by static analysis, therefore messages 4113 (The declaration of object '%1s' can be moved to this nested scope) and 4114 (The declaration of the constant object '%1s' can be moved to this nested scope) can be seen as false positives that are not desirable.

Example:

```
#define X int i; const int j ; \
          if ( true ) { i += j ; }

void bar ()
{
  X;
}
```

Resolution:

This issue has been fixed in the QA-C++ parser.

4.2 False positive for a call to a non-static member template function of the enclosing class in a lambda expression if it has explicitly specified template arguments.

Problem Description:

Using a template-id to call a non-static member template function of the enclosing class from a lambda expression resulted in a false positive message 406.

Example:

```
struct A
{
  void foo()
  {
    [this] () { bar<char>(); };
  }

  template<typename T>
  void bar();
};
```

Resolution:

This issue has been fixed in the QA-C++ parser.

Chapter 5

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++'14 with some implementation of support for C++'17 versions of the language and created the new compliance module for RePhrase standard. This implementation have made PRL tool to be able to analyse comprehensive parallelisation code and provide diagnostics for violation of new standard. During recent trip to USA we have found a large interest to this standard and compliance module from the few large customers.