Project no. 644235

REPHRASE

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

Final report on data and coding standard
D5.5

Due date of deliverable: 30.06.2017

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP5

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

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)	

Change Log

Rev.	Date	Who	Site	What
1	12/05/17	Evgueni Kolossov	PRQA	Created original draft document
2	9/06/17	Evgueni Kolossov	PRQA	Incorporated feedback and revised document
3	26/06/17	Evgueni Kolossov	PRQA	Finalised document

Executive Summary

This document is the deliverable of WP5 "Integration and Overall Software Engineering Methodology", and in particular Task 5.3 "Data and Coding Standards".

It describes the final rule set of the RePhrase C++ Coding Standard that facilitates parallel data-intensive software engineering. As shown in Figure below, this coding standard is based on hygienic parallel properties identified in deliverables D2.1 "Report on the initial pattern set", D2.3 "Report on shaping and pattern discovery for initial patterns", and D5.5 "Refined report on data and coding standard" as well as drawing on pre-existing guidance of other coding standards and publications.

This deliverable will provide direct input to WP3 "Reliability, Robustness and Software Integrity of Parallel Software", Task 3.4 "Quality Assurance Analysis", as well as impacting deliverable D2.9 "Report on shaping and pattern discovery for advanced patterns" and D3.4 "Software for the final version of QA tool".

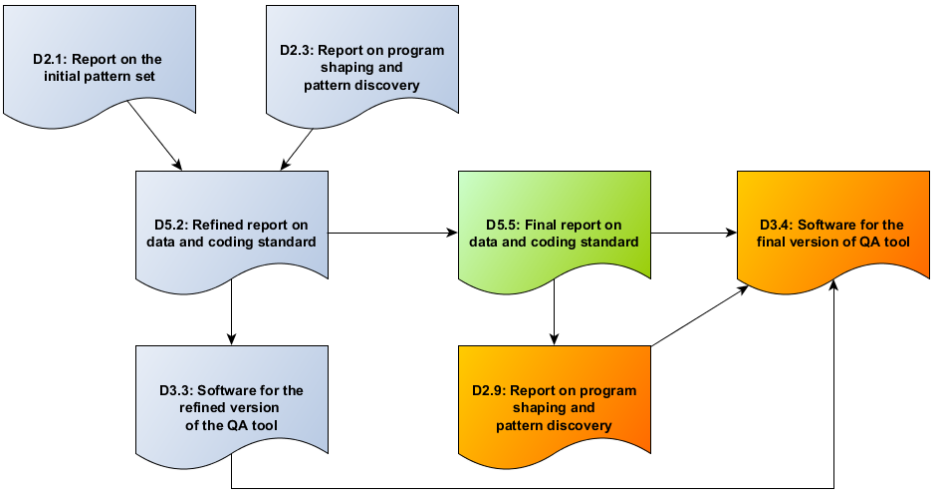

Figure 1: Dependencies of D5.5

Contents

1. Introduction

The C++ language is rapidly changing and moving forward. A significant number of new features relate to multi-threading and parallel programming.

The plethora of multi-core and many-core architectures means that making use of parallel language features is important in order to achieve maximum hardware utilisation. It is only with recent versions of C++ that standard library features have become available to make use of this hardware in a standard compliant way. As with many features of C++ appropriate coding guidelines can help avoid common pitfalls in their use.

The RePhrase project aims to make use of the functionality for multi-threading and parallel programming through the use of patterns. These patterns may also have common pitfalls which appropriate guidelines can help mitigate against. The goal for this delivery is to provide the final set of guidelines based on the review and feedback received on the initial beta version (D.52). The set of guidelines will take the following sources into consideration:

- Output from D2.1 and D2.3
- The HIC++ Coding Standard [3]
- The ISO C++ Standard [2]
- Feedback on D5.2

1.1 Document Layout

Throughout this document, a rule is formatted using the following structure.

Justification:	Text that provides rationale and example(s). Examples shall, where possible, include both *Compliant* and *Non-compliant* samples.
Exception:	This paragraph explains cases where the rule does not apply.

This standard contains many example code fragments which are designed to illustrate the meaning of the rules. For example, the following is an example of a hypothetical rule that requires all variables to have an explicit initializer:

```
void foo ()
{
  int i1;      // Non-Compliant
  int i2 = 0; // Compliant
}
```

For cases where the significant example spans multiple lines, or where the comment will not fit on the end of the line, the *Compliant* and *Non-Compliant* will be placed before the construct:

```
void foo ()
{
  // Non-Compliant: Variable 'i1' is not initialised
  int i1;

}
```

For brevity some of the example code does not conform to all best practises, e.g. unless the rule relates explicitly to concurrency, the example code may not be thread-safe.

2 General Rules

In a sequential program a minor use of *unspecified*, *implementation defined* or *undefined behavior* may well go unnoticed; however, it is much more likely to impact a non-sequential program.

The following section specifies rules that aim to minimise, if not remove completely non deterministic behaviors that decrease the overall hygiene of the code. These rules are of benefit to both sequential and non-sequential programs.

2.1 Ensure that all statements are reachable

Justification:
For the purposes of this rule missing `else` and `default` clauses are considered also.
If a statement cannot be reached for any combination of function inputs (e.g. function arguments, global variables, volatile objects), it can be eliminated.
For example, when the condition of an if statement is never false the `else` clause is unreachable. The entire `if` statement can be replaced with its 'true' sub-statement only.
In practice two methods are used to detect unreachable code:

- sparse conditional constant propagation
- theorem proving

by showing that non-execution of a statement is independent of function inputs. Since the values of variables are not used to determine unreachability, this restricted definition is decidable.
A compiler may detect and silently remove unreachable statements as part of its optimizations. However, explicitly removing unreachable code has other benefits apart from efficiency: the structure of the function will be simplified. This will improve its maintainability and will increase the proportion of statements that can be reached through coverage analysis.
For Example:

```
bool f1 (int a) {
  return true;
}

void f2 (int b) {
  // Non-Compliant: implicit else clause cannot be reached
  //                for any 'b'
  if (f1 (b)) {
    // ...
  }

  f1 (b); // Compliant
}
```

2.2 Ensure that pointer or array access is demonstrably within bounds of a valid object

Justification:
Unlike standard library containers, arrays do not benefit from bounds checking. Array access can take one of the equivalent forms: `*(p + i)` or `p[i]`, and will result in undefined behavior, unless `p` and `p + i` point to elements of the same array object. Calculating (but not dereferencing) an address one past the last element of the array is well defined also. Note that a scalar object can be considered as equivalent to an array dimensioned to 1.
To avoid undefined behavior, appropriate safeguards should be coded explicitly (or instrumented by a tool), to ensure that array access is within bounds, and that indirection operations ($\star$) will not result in a null pointer dereference.
For Example:

```
#include <cassert>

void foo (int* i)
{
  int k = *i;           // Non-Compliant: could be nullptr

  assert (i != nullptr);
  k = *i;               // Compliant

  int a [10];
  for (int i (0); i < 10; ++i)
  {
    a [i] = i;          // Compliant, array index is 0..9
  }

  int * p = & (a [10]); // Compliant: one past the end
  k = *p;               // Non-Compliant: out of bounds
}
```

2.3 Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero

Justification:
The result of integer division or remainder operation is undefined if the right hand operand is zero. Therefore, appropriate safeguards should be coded explicitly (or instrumented by a tool) to ensure that division by zero does not occur.
For Example:

```
#include <cassert>

int doDivide1(int number, int divisor)
{
  return number / divisor;  // Non-Compliant
}

int doDivide2(int number, int divisor)
{
  assert (0 != divisor);
  return number / divisor;  // Compliant
}
```

2.4 Do not return a reference or a pointer to an automatic variable defined within the function

Justification:
The lifetime of a variable with automatic storage duration ends on exiting the enclosing block. If a reference or a pointer to such a variable is returned from a function, the lifetime of the variable will have ended before the caller can access it through the returned handle, resulting in undefined behavior.
For Example:

```
class String
{
public:
  String (char *);
  String (const String &);
};

String & fn1 (char * myArg)
{
  String temp (myArg);
  return temp;          // Non-Compliant: temp destroyed here
}

String fn2 (char * myArg)
{
  String temp (myArg);
  return temp;          // Compliant: the caller will get a
                        //            copy of temp
}
```

2.5 Do not assign the address of a variable to a pointer with a greater lifetime

Justification:
The C++ Standard defines 4 kinds of storage duration:

- static
- thread
- automatic
- dynamic

The lifetime of objects with the first 3 kinds of storage duration is fixed, respectively:

- until program termination
- until thread termination
- upon exiting the enclosing block

Therefore, undefined behavior will likely occur if an address of a variable with automatic storage duration is assigned to a pointer with static or thread storage duration, or one defined in an outer block. Similarly, for a `thread_local` variable aliased to a pointer with static storage duration.
For Example:

```
void foo (bool b)
{
  int * p;
  if (b)
  {
    int c = 0;
    int * q = &c;  // Compliant
    p = &c;        // Non-Compliant
  }
}
```

2.6 Do not code side effects into the predicate argument to `condition_variable::wait`, the argument to `assert` or the right-hand operands of: `&&`, `||`, `sizeof`, `typeid`

Justification:
For some expressions, the side effect of a sub-expression may not be evaluated at all or can be conditionally evaluated, that is, evaluated only when certain conditions are met. For Example:

- The right-hand operands of the `&&` and `||` operators are only evaluated if the left hand operand is `true` and `false` respectively.
- The operand of `sizeof` is never evaluated.
- The operand of `typeid` is evaluated only if it is a function call that returns reference to a polymorphic type.
- The argument to `assert` is evaluated only when `NDEBUG` is not defined.
- The predicate argument to `condition_variable::wait` may be evaluated without notification.

Having visible side effects that do not take place, or only take place under special circumstances makes the code harder to maintain and can also make it harder to achieve a high level of test coverage.
For Example:

```
#include <typeinfo>

bool doSideEff();

class C
{
public:
  virtual ~C(); // polymorphic class
};

C& f1();

void f2( bool condition )
{
  if (doSideEff () && condition) // Compliant
  {}

  if (false && doSideEff ()) // Non-Compliant: not called
  {}

  if (true || doSideEff ())  // Non-Compliant: not called
  {}

  sizeof (doSideEff()); // Non-Compliant: not called
  typeid (doSideEff()); // Non-Compliant: not called
  typeid (f1());        // Non-Compliant: f1 called to find
                        //                polymorphic type
}
```

Conditional Variable: Every time a waiting thread wakes up, it checks the predicate passed to the `wait` member. The wake-up may not necessarily happen in

direct response to a notification from another thread. This is called a spurious wake. It is indeterminate how many times and when such spurious wakes happen. Therefore it is advisable to avoid using a function with side effects to perform the condition check.

For Example:

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

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

int i;

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

void threadX()
{
  i = 0;
  std::unique_lock<std::mutex> guard(mut);
  cv.wait(guard, sideEffects);   // Non-Compliant
                                 // value of i depends on
                                 // the number of wakes
  cv.wait(guard, noSideEffects); // Compliant
}
```

2.7 Implement a loop that only uses element values using range-for or an STL algorithm

Justification:
A range-based for statement reduces the amount of boilerplate code required to maintain correct loop semantics.
A range-based loop can normally replace an explicit loop where the index or iterator is only used for accessing the container value.
For Example:

```
#include <iterator>

void bar ()
{
  int array[] = { 0, 1, 2, 3, 4, 5, 6 };
  int sum = 0;

  //  Non-Compliant
  for (const int * p = std::cbegin (array)
       ; p != std::cend (array)
       ; ++p)
  {
    sum += *p;
  }

  sum = 0;
  // Compliant
  for (int v : array )
  {
    sum += v;
  }

  // Compliant
  for (size_t i = 0
       ; i != (sizeof(array)/sizeof(*array))
       ; ++i)
  {
    if ((i % 2) == 0)        // Using the loop index
    {
      sum += array[i];
    }
  }
}
```

2.8 Ensure that execution of a function with a non-void return type ends in a return statement with a value

Justification:
Undefined behavior will occur if execution of a function with a non void return type (other than `main`) flows off the end of the function without encountering a return statement with a value.
For Example:

```
int f1 (bool b)
{
  if (b) {
    return -1;
  }
  // Non-Compliant: flows off the end of the function
}

int f2 (bool b)
{
  if (b) {
    return -1;
  }
  return 0; // Compliant
}
```

Exception:
The `main` function is exempt from this rule, as an implicit `return 0;` will be executed, when an explicit `return` statement is missing.

2.9 Postpone variable definitions as long as possible

Justification:
To preserve locality of reference, variables with automatic storage duration should be defined just before they are needed, preferably with an initializer, and in the smallest block containing all the uses of the variable.
For Example:

```
int f1 (int v)
{
  int i;          // Non-Compliant

  if ((v > 0) && (v < 10))
  {
    i = v * v;
    --i;
    return i;
  }
  return 0;
}

int f2 (int v)
{
  if ((v > 0) && (v < 10))
  {
    int i (v*v); // Compliant
    --i;
    return i;
  }
  return 0;
}
```

The scope of a variable declared in a for loop initialisation statement extends only to the complete for statement. Therefore, potential use of a control variable outside of the loop is naturally avoided.
For Example:

```
int f3 (int max)
{
  int i;
  for (i = 0; i < max; ++i)      // Non-Compliant
  {
  }
  return i;
}

void f4 (int max)
{
  for (int i (0); i < max; ++i) // Compliant
  {
  }
}
```

2.10 Use `const` whenever possible

Justification:
This allows specification of semantic constraint which a compiler can enforce. It explicitly communicates to other programmers that value should remain invariant. For example, specify whether a pointer itself is const, the data it points to is const, both or neither.
For Example:

```
struct S
{
  char* p1;             // non-const pointer to non-const
  const char* p2;       // non-const pointer to const
  char* const p3;       // const pointer to non-const
  const char* const p4; // const pointer to const
};

void foo (const char * const p); // Compliant

void bar (S & s) // Non-Compliant: parameter could be
                 //                const qualified
{
  foo (s.p1);
  foo (s.p2);
  foo (s.p3);
  foo (s.p4);
}
```

Exception:
By-value return types are exempt from this rule. These should not be `const` as doing so will inhibit move semantics.
For Example:

```
struct A { };

const int f1 ();   // Non-Compliant
const A f2 ()  ;   // Non-Compliant
A f3 ();           // Compliant
```

2.11 Do not use the `asm` declaration

Justification:
Use of inline assembly should be avoided since it restricts the portability of the code.
For Example:

```
int foo ()
{
  int result;

  asm ("");     // Non-Complaint

  return result;
}
```

2.12 Do not access an invalid object or an object with indeterminate value

Justification:
A significant component of program correctness is that the program behavior should be deterministic. That is, given the same input and conditions the program will produce the same set of results.
One category of behaviors affecting determinism constitutes use of:

- variables not yet initialized
- memory (or pointers to memory) that has been freed
- moved from objects

Note: There may be other causes of non-deterministic behaviour, such as presence of undefined behavior or violations of rules 3.4 or 3.5.
For Example:

```
#include <iostream>

class A
{
public:
  A();
  // ...
};

std::ostream operator<<(std::ostream &, A const &);

int main ()
{
  int i;
  A a;

  // Non-Compliant: 'i' has indeterminate value
  std::cout << i << std::endl;

  // Compliant: Initialized by constructor call
  std::cout << a << std::endl;
  return 0;
}
```

Note: For the purposes of this rule, after the call to `std::move` has been evaluated the moved from argument is considered to have an indeterminate value.
For Example:

```
#include <vector>

int main ()
{
  std::vector<int> v1;
  std::vector<int> v2;

  std::vector<int> v3 (std::move (v1));
  std::vector<int> v4 (std::move (v2));

  v1.empty (); // Non-Compliant: 'v1' considered to have
               //                indeterminate value

  v2 = v4;     // Compliant: New value assigned to 'v2'
  v2.empty (); //            before it is accessed    '
  return 0;
}
```

2.13 Declare `static` any member function that does not require `this`. Alternatively, declare `const` any member function that does not modify the externally visible state of the object

Justification:
A non-virtual member function that does not access the `this` pointer can be declared `static`. Otherwise, a function that is non-virtual and does not modify the externally visible state of the object can be declared `const`.
Note: Virtual functions may also be made const, however, this requires that none of the overrides modify the externally visible state of the object..
The `C++` language permits that a const member function modifies the program state (e.g. modifies a global variable, or calls a function that does so). However, it is recommended that const member functions are logically const also, and do not cause any side effects.
The `mutable` keyword can be used to declare member data that can be modified in a const function, however, this should only be used where the member data does not affect the externally visible state of the object.

For Example:

```
struct C
{
  explicit C (int i) : m_i (i) , m_c (0) { }

  int f1 () { // Non-Compliant: should be static
    C tmp (0);
    return tmp.f2 ();
  }

  int f2 () { // Non-Compliant: should be const
    ++ m_c;
    return m_i;
  }

  static int f3 () { // Compliant
    return C(0).f2 ();
  }

private:
  int m_i;
  mutable int m_c;
};
```

2.14 Do not return non-const handles to class data from const member functions

Justification:
A pointer or reference to non-const data returned from a const member function may allow the caller to modify the state of the object. This contradicts the intent of a const member function.
For Example:

```
class C
{
public:
  C () : m_i (new int) {}

  ~C() { delete m_i; }

  int * get () const {
    return m_i;  // Non-Compliant
  }

  int const * getc () const {
    return m_i;  // Compliant
  }

private:
  int * m_i;

  C (C const &) = delete;
  C & operator = (C const &) & = delete;
};
```

Exception:
Resource handler classes that do not maintain ownership of a resource are exempt from this rule.
For Example:

```
class D
{
public:
  D (int * p) : m_i (p) {}

  int * get () const
  {
    return m_i;  // Compliant
  }

private:
  int * m_i;
};
```

2.15 Ensure that `std::terminate` demonstrably is not called

Justification:
A call to `std::terminate` will cause the program to immediately stop, potentially leaving corrupted data and state.
For Example:

```
#include <fstream>

void bar () noexcept
{
  throw 0; // calls std::terminate
}

int main ()
{
  std::ofstream fout("my.cfg");
  fout << "Start\n";
  bar ();
  fout << "End" << std::endl;
}
```

3 Concurrency Rules

Programs that make use of multi-core and many-core architectures will require at least some of the data to be read and/or written to from sections of code running in parallel. Of particular concern is accessing resources that are written to or read from code that is not running sequentially. One of the most important and interesting features in recent C++ Standards are concurrency primitives which enable the synchronisation of access to resources that are shared between code that is running in parallel.

This section provides guidelines for the correct use of the new language features that provide these synchronisation primitives provided by the recent C++ Standard.

3.1 Do not use platform specific multi-threading facilities

Justification:
Rather than using platform-specific facilities, the C++ standard library should be used as it is platform independent.
For 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);
  // ...
}
```

Where the standard library does not offer all the required facilities, e.g. for setting thread priority, localised use of `std::thread::native_handle` with a platform specific API is preferred.
For Example:

```
#include <pthread.h>
#include <thread>
void threadFunc();

int setThreadPriority(std::thread & t, int priority)
{
  sched_param param;
  int policy;
// Permitted exception
  pthread_getschedparam(t.native_handle(), &policy, &param);
  param.sched_priority = priority;
  return pthread_setschedparam(t.native_handle(), policy, &param);
}

void foo()
{
  std::thread t(threadFunc);

  setThreadPriority (t, 20);

  // ...
}
```

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

Justification:
`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.
For 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);
}
```

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

Justification:
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.
For 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);
}
```

3.4 Synchronise access to data shared between threads using the same lock

Justification:
Using the same lock when accessing shared data makes it easier to verify the absence of problematic race conditions.
To help achieve this goal, access to data should be encapsulated such that it is not possible to read or write to the variable without acquiring the appropriate lock. This will also help limit the amount of code executed in the scope of the lock.
Note: Data may be referenced by more than one variable, therefore this requirement applies to the complete set of variables that could refer to the data.

For Example:

```
#include <mutex>
#include <string>

class some_data {
public:
  void do_something();

private:
  int a;
  std::string b;
};

some_data* unprotected;

void malicious_function(some_data& protected_data) {
  // Suspicious, unprotected now refers
  // to data protected by a mutex
  unprotected=&protected_data;
}

class data_wrapper
{
public:
  template<typename Function>
  void process_data(Function func) {
    std::lock_guard<std::mutex> lk(m);
    func(data);          // 'protected_data' assigned to
                         // 'unprotected' here
  }

private:
  some_data data;
  mutable std::mutex m;
};

data_wrapper x;

void foo() {
  x.process_data(malicious_function);

  // Not Compliant: 'unprotected' accessed
  //                outside of 'data_wrapper::m'
  unprotected->do_something();
}
```

3.5 Access to `mutable` members shall be synchronised in const member functions, for an object shared between threads

Justification:
The standard library expects operations on const objects to be thread-safe. Failing to ensure that this expectation is fulfilled may lead to problematic data races and undefined behavior. Therefore, operations on const objects of user defined types should consist of either reads entirely or internally synchronized writes.
For Example:

```
#include <mutex>
#include <atomic>
#include <future>

class A
{
public:
  int get1() const
  {
    ++counter1;    // Non-Compliant: unsynchronized write to
                   //                a data member of non
                   //                atomic type

    ++counter2;    // Compliant: write to a data member of
                   //            atomic type
  }
  int get2() const
  {
    std::lock_guard<std::mutex> guard(mut);
    ++counter1;    // Compliant: synchronised write to data
                   // member of non atomic type
  }
private:
  mutable std::mutex           mut;
  mutable int              counter1;
  mutable std::atomic<int> counter2;
};

void worker(A & a);
void foo(A & a)
{
  auto f = std::async (std::launch::async, worker, std::ref (a));
}
```

3.6 Access to volatile data shared between threads shall have appropriate synchronisation

Justification:
Declaring a variable with the *volatile* keyword does not provide any of the required synchronization guarantees:

- atomicity
- visibility
- ordering

For Example:

```
#include <functional>
#include <future>
#include <thread>
#include <chrono>

using namespace std::chrono_literals;

// Non-Compliant - using volatile for synchronisation
class DataWrapper {
public:
  DataWrapper () : flag (false), data (0){}

  void incrementData() {
    while(flag) {
      std::this_thread::sleep_for(1s);
    }
    flag = true;
    ++data;
    flag = false;
  }

  int getData() const {
    while(flag) {
      std::this_thread::sleep_for(1s);
    }
    flag = true;
    int result (data);
    flag = false;

    return result;
  }

private:
  mutable volatile bool flag;
  int data;
};

void worker(DataWrapper & data);
void foo(DataWrapper & data) {
  auto f = std::async (std::launch::async, worker, std::ref (data));
}
```

Use mutex locks or ordered atomic variables, to safely communicate between threads and to prevent the compiler from optimising the code incorrectly.

For Example:

```
#include <functional>
#include <mutex>
#include <future>

// Compliant - using locks
class DataWrapper {
public:
  DataWrapper () : data (0) { }
```

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

  int getData() const  {
    std::lock_guard<std::mutex> guard(mut);
    return data;
  }

private:
  mutable std::mutex mut;
  int data;
};

void worker(DataWrapper & data);
void foo(DataWrapper & data)
{
  auto f = std::async (std::launch::async, worker, std::ref (data));
}
```

3.7 Shared global data shall be provided through static local objects

Justification:
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.
For 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;
}
```

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

Justification:
The standard library provides the `std::call_once` that can be used to guarantee that a call is made at most once.
For 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 World");
  // ...
}
```

When used correctly, patterns such as the Double-Checked Locking Pattern provide similar synchronisation, however, providing a correct implementation is not trivial so their use is not recommended.

3.9 Use a `noexcept` lambda as the argument to a thread constructor

Justification:
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.
For 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 ();
}
```

The lambda shall be declared `noexcept` to explicitly document that for a thread exiting via an exception `std::terminate` will be called, see 2.15. The intention is that catch handlers, alternatively, manual or automated analysis will be used to ensure exceptions do not propagate out of the lambda/thread.

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

Justification:
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.
For Example:

```
#include <thread>

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

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

Justification:
It is undefined behavior if a thread tries to lock a std::mutex it already owns, this should therefore be avoided.
For 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 from f1
    std::lock_guard<std::mutex> hold(mut);
    ++i;
  }
}
```

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

Justification:
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).
For Example:

```
#include <mutex>

// Non-Compliant:  Nesting of locks does not form a 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;
};
```

The corrected example is as follows:

```
#include <mutex>

// Compliant:  Nesting of locks forms a DAG:
//             mut1->mut2 and then mut1->mut2
class B
{
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> lock1(mut1);
    std::lock_guard<std::mutex> lock2(mut2);
    ++i;
  }

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

3.13 Do not use `std::recursive_mutex`

Justification:
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.
For 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;
};
```

Such situations should be solved by redesigning the code.
For Example:

```
// Compliant: Not using mutex
#include <mutex>

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

  void incrementData() {
    std::lock_guard<std::mutex> guard(mut);
    inc();
  }

  // ...
private:
  void inc() {
    // expects that the mutex has already been locked
    ++data;
  }

  mutable std::mutex mut;
  int data;
};
```

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

Justification:
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`.
For 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);  // Compliant
  // critical code here
  // lock automatically released
}
```

Exception:
Where a conditional variable is required, it is acceptable to use `std::unique_lock` with `std::conditional_variable`.
For Example:

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

std::mutex mut;
std::condition_variable 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);

  // Compliant
  cv.wait(guard, []{ return !container.empty(); } );

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

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

Justification:
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.
For Example:

```
#include <mutex>

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

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

3.16 Do not use relaxed atomics

Justification:
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.
For 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) == 1) {
      delete m_ptr;
    }
  }

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

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

Justification:
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.
For 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();
}
```

3.18 Do not unlock a mutex which is not already held by the current thread

Justification:
Unlocking a mutex which is not already held by the current thread results in undefined behaviour.
For Example:

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

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

void tryPush(int i)
{
  if (mut.try_lock())
  {
    container.push_back(i);
  }

  mut.unlock();  // Non Compliant: mut not always held by current thread
}
```

3.19 Do not destroy a locked mutex

Justification:
Destroying a mutex which is locked results in undefined behaviour.
For Example:

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

struct A
{
  A()
  { }

  void clear()
  {
    m_mut.lock();
    m_v.clear();
  }

 private:
  mutable std::mutex m_mut;
  std::vector<int> m_v;
};

void foo()
{
  A a;
  a.clear();
  // Non Compliant: a.m_mut still locked when destructor is called
}
```

3.20 Before exiting a thread ensure all mutexes held by it are unlocked

Justification:
As mutexes can only be unlocked by the thread holding the mutex, any mutexes still being locked when a thread exists can never be unlocked and could result in a deadlock if any other thread tries to acquire one of these mutexes.
For Example:

```
#include <mutex>
#include <vector>
#include <thread>

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

void tryPush(int i)
{
  if (mut.try_lock())
  {
    container.push_back(i);
  }
}

void foo()
{
  // Non Compliant: the thread may still hold mut when exiting
  std::thread t(tryPush, 1);
  t.detach();
}
```

3.21 Atomics used in a signal handler shall be lock-free

Justification:
Atomic types are not always lock-free. They can be implemented using mutexes in which case they are no longer signal-safe. Code using atomics in a signal handler should therefore ensure that the atomics are lock-free to avoid undefined behaviour. In C++ '17 this can be done checking `is_always_lock_free` in a `static_assert`, but in older versions of C++ this can only be achieved via `ATOMIC_*_LOCK_FREE` macros.
For Example:

```
#include <atomic>

std::atomic<long> value;

void handler(int signum)
{
  // Compliant: ensure that the atomic type is always lock-free
  static_assert (ATOMIC_LONG_LOCK_FREE == 2, "atomic␣is␣not␣always␣lock-free");

  // in C++ '17 is_always_lock_free() can be checked instead
  static_assert (value.is_always_lock_free(), "atomic␣is␣not␣always␣lock-free");

  ++value;
}
```

3.22 Do not use an object with thread storage duration in a signal handler

Justification:
Accessing objects with thread storage duration can result in implicitly generated calls to non-signal-safe functions. It is undefined-behavior to call a non-signal-safe function from a signal handler.
For Example:

```
thread_local int value;

void handler(int signum)
{
  // Non-Compliant: accessing an object with thread storage duration
  ++value;
}
```

4 Parallelism Rules

The C++ Standard Template Library (STL) provides a set of common classes, interfaces and algorithms that greatly extend the core C++ language. Until recently, neither the core language nor the STL considered code running non-sequentially. The result being that in order to make use of multi-core and many-core architectures developers were required to use platform specific tools and libraries.

To address this, the ISO Committee worked on the Parallelism Technical Specification [1] which, at time of writing, has been integrated into the version of ISO C++ 2017 that will be sent for ballot. The result is an expanded STL with parallel versions of pre-existing algorithms and the addition of new STL algorithms for some of the parallel patterns described in D2.3 "Report on shaping and pattern discovery for initial patterns".

This section provides guidelines for the correct use of these new parallelism features.

4.1 Use higher-level standard facilities to implement parallelism

Justification:
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.

4.2 Functor used with a parallel algorithm shall be pure

Justification:
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.

For Example:

```
int * f2(int i, int j)
{
  // Compliant
  int * k = new int (i + j);
  return k;
}
```

For determining if a function is pure or not, taking the address of a variable will be considered as a read of the variable itself. For example, taking the address of a variable with static storage duration will result in the function being non-pure:
For Example:

```
int m;
int * f2()
{
  // Non-compliant: 'reads' m
  return &m;
}
```

4.3 Loops and Functors should not exhibit unhygienic properties

Justification:
Only loops and functors that don't exhibit any unhygienic properties can easily be parallelised. Unhygienic properties of loops are:

- Accessing objects visible outside of the loop
- Jumps
- Throw
- Calling non-pure functions 4.2

Any attempt to parallelise a loop exhibiting one of the unhygienic properties can result in seemingly random and hard to detect problems at runtime.

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

Justification:
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.

For 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;
}
```

4.5 Functor used with a parallel algorithm shall always return

Justification:
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

For 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) ;
  }
};
```

Exception:
When determining if a function returns, exceptions thrown and the assert macro are not considered:
For Example:

```
#include <cassert>

class DoSomething {
  // Compliant
  void operator()(int * i) {
    assert ( i && "Must␣have␣valid␣object" );
    // ...
  }
};
```

4.6 Functors used with parallel algorithms shall be `noexcept`

Justification:
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.
For 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)
  {
  }
}
```

For Example:

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

void f(std::vector<int> & v)
{
  // Compliant
  std::for_each (std::execution::par
    , v.begin ()
    , v.end ()
    , [](int) noexcept { } );
}
```

See also 2.15.

4.7 Catch handlers enclosing algorithms with execution policies shall include `std::bad_alloc`

Justification:
An algorithm called with an execution policy may fail due to lack of memory. In such cases, an exception of type `std::bad_alloc` will be thrown.
For Example:

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

void f1 (std::vector<int> & v)
{
  // Non Compliant
  try
  {
    std::for_each (std::execution::par
      , v.begin ()
      , v.end ()
      , [](int) noexcept { /* ... */ } );
  }
  catch (int & e)
  {
  }
}

void f2 (std::vector<int> & v)
{
  // Compliant
  try
  {
    std::for_each (std::execution::par
      , v.begin ()
      , v.end ()
      , [](int) noexcept { /* ... */ } );
  }
  catch (std::bad_alloc & e)
  {
  }
}
```

Note: A catch all handler, `catch(...)`, is also acceptable.

4.8 The `binary_op` used with `std::reduce` or `std::transform_reduce` shall be demonstrably associative and commutative

Justification:
The behavior of these algorithms is non deterministic when `binary_op` is not associative or commutative.
For Example:

```
#include <numeric>
#include <string>

int main()
{
  char carr[] = { 'H', 'e', 'l', 'l', 'o' };
  // Non Compliant
  auto s = std::reduce (std::execution::par
    , std::cbegin (carr)
    , std::cend (carr)
    , std::string ()
    , [] (auto const & res, auto elem) { return res + elem; });

  int iarr[] = { 1, 2, 3, 4, 5 };
  // Compliant
  auto v = std::reduce (std::execution::par
    , std::cbegin (iarr)
    , std::cend (iarr)
    , 0
    , [] (auto val, auto elem) { return val + elem; });
}
```

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

Justification:
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.
For 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 (); });
}
```

5 Conclusion

The goal of this deliverable was to provide the final set of coding standard rules as part of Task 5.3 "Data and Coding Standards". For this, the initial output from D2.1, D2.3, D5.2, coding standards such as HIC++ as well as a review of recent ISO C++ Standard drafts were considered.

The result was the expansion of an initial set of 42 rules into 46 rules in 3 categories:

- 15 General Rules,
- 22 Concurrency Rules,
- 9 Parallelism Rules

The set of General Rules can be applied to both sequential and non-sequential programs, and aim to improve the determinism of programs at runtime. The current set of Concurrency Rules address the correct usage of synchronisation features provided by the standard C++ language, while the set of Parallelism Rules covers the use of the parallel versions of the C++ standard library algorithms.
These rules can be applied to:

- non-sequential code, where they help optimise the refactoring and parallelisation of that code
- parallel code, where they help detect problems that may lead to non-deterministic behaviour of the resulting program.

The final set of rules defined here will be an input for D2.9 "Report on program shaping and pattern discovery" and will lead to creation of the compliance module for the final version of PRL QA-C++ software in D3.4: "Software for the final version of QA tool"

Bibliography

[1] ISO/IEC. Programming languages - technical specification for c++ extensions for parallelism, 2015.

[2] ISO/IEC. Working draft, standard for programming language c++, 2016.

[3] Programming Research Ltd. High integrity c++ coding stand, 2013.