Project no. 644235

REPHRASE

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

Initial Report on Evaluation for Reliability, etc.
D6.5

Due date of deliverable: 30.06.2017

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP6

Responsible institution: PRL
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	2/06/17	Evgueni Kolossov	PRQA	Created original document
2	20/06/17	Evgueni Kolossov	PRQA	Incorporated feedback and revised document
3	29/06/17	Michael Rossbory	SCCH	Minor Improvements

Executive Summary

This deliverable is the report on initial evaluation of the applicability of Rephrase methodology and tools for the test projects ported by other partners in D6.2, with respect to Rephrase code quality goals of reliability, robustness, resilience, integrity and adaptivity as described in D.6.2.

We use the PRL QA-Verify tool to evaluate the initial use case code, thus deriving an initial baseline. A final analysis takes place after the Rephrase approach has been applied, and is compared to the initial baseline. We compare the quality of the use case before and after application of the Rephrase approach, using the metrics that are developed in D6.1 to cover appropriate impact of the Rephrase methodology and tools. This is the initial evaluation. In D6.7 a final report will be delivered on evaluation of the use cases and methodology for these software quality criteria, taking into account the updated applications in D6.6.

The relationships with other deliverables shown in the Figure below:

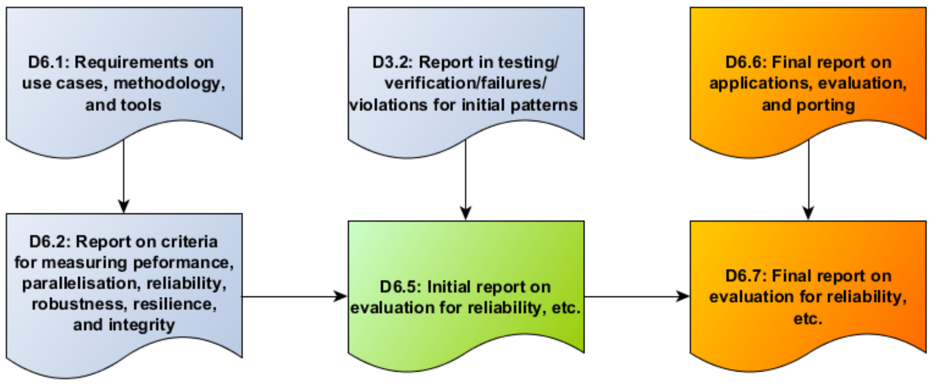

D6.1: Requirements on use cases, methodology, and tools
D3.2: Report in testing/ verification/failures/ violations for initial patterns
D6.6: Final report on applications, evaluation, and porting
D6.2: Report on criteria for measuring peformance, parallelisation, reliability, robustness, resilience, and integrity
D6.5: Initial report on evaluation for reliability, etc.
D6.7: Final report on evaluation for reliability, etc.

Figure 1: Dependencies of D6.5

Contents

Chapter 1

Introduction

As mentioned in D6.2 report, the main issue with the metrics mentioned in the Executive Summary is the lack of knowledge about how to measure them via automated static analysis (ASA) tools:

- Reliability + Robustness + Adaptivity, mean time to failure (MTTF), and mean time between Failures (MTBF).

- Resilience: validation correct and continued operation under injected problems, based on modelling the kind of problems the system is expected to endure.

- Integrity (ratio between covered and total code elements): needs to be measured via integration with code coverage tools.

The evaluation is also a challenge to runtime analysis. Based on strict definitions it depends on failures being recorded, and doing so within a bounded and practical amount of time. MTBF also depend on mean time to repair (MTTR) whose applicability in software adds additional constraints/definitions which may not be applicable for our use cases. There is limited uptake and agreement on such evaluation in commercial software products. The Software Assurance Technology Centre (SATC) at NASA defines Reliability as a "Quality attribute" (i.e. likelihood of failure) and "analyzes the code for the structure and architecture to identify possible error prone modules based on complexity, size, and modularity". It relies mainly on requirement analysis, static analysis metrics similar to those produced by PRQA tools, and evaluation through testing of the number of errors in code and rate of finding/fixing them - see Software Metrics and Reliability.

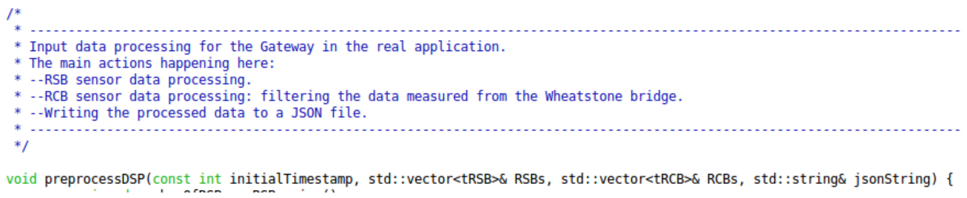
In the context of evaluating pre and post-refactoring of the use cases, the approach taken is likewise to rely on static analysis metrics exposed in D6.2 and compound metrics based on them, in order to monitor the impact of the Rephrase methodology on quality.

A mechanism is also described for upload of new metrics to the QA-Verify tool for convenient comparison between snapshots. Some performance metrics are provided on the Railway use case, for which PRL holds enough data to run a CPU-based binary. Additional metrics obtained from tools owned by the partners and/or obtained from Rephrase tools such as the PAPI-based tool described in the D3.2 report (which deals with detection of violations of extra-functional properties, such as performance or energy consumption) can be uploaded in the format specified in this document to enhance the evaluation.

Chapter 2

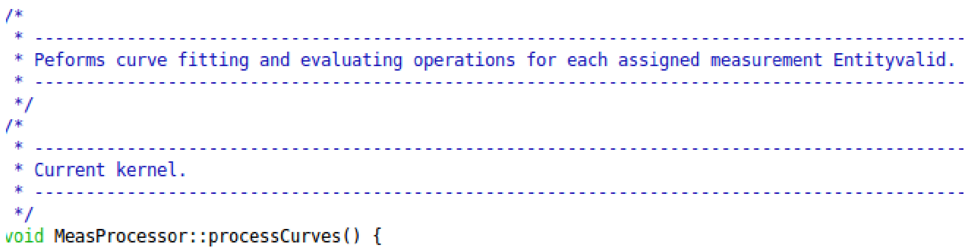
Static Analysis Metrics

The metric result exposed from the Railway use case on the CPU versions with sequential and parallel code(FastFlow). We are looking at 2 functions that have been parallelised:



```
/*
 * ------------------------------------------------------------------------------------------------------------------------
 * Input data processing for the Gateway in the real application.
 * The main actions happening here:
 * --RSB sensor data processing.
 * --RCB sensor data processing: filtering the data measured from the Wheatstone bridge.
 * --Writing the processed data to a JSON file.
 * ------------------------------------------------------------------------------------------------------------------------
 */

void preprocessDSP(const int initialTimestamp, std::vector<tRSB>& RSBs, std::vector<tRCB>& RCBs, std::string& jsonString) {
```

Figure 2.1: preprocessDSP function



```
/*
 * ------------------------------------------------------------------------------------------
 * Peforms curve fitting and evaluating operations for each assigned measurement Entityvalid.
 * ------------------------------------------------------------------------------------------
 */
/*
 * ------------------------------------------------------------------------------------------
 * Current kernel.
 * ------------------------------------------------------------------------------------------
 */
void MeasProcessor::processCurves() {
```

Figure 2.2: processCurves function

2.1 Reliability

The term "Reliability" is meant to also cover Robustness and Resilience, since as discussed above these metrics cannot be discriminated through static analysis alone, and the three of them contribute to the quality attribute.

2.1.1 Knots

Knot count metric is implemented inside PRL software (A.2.13 STKNT: Knot Count). This is the number of knots in a function. This metric measures the complexity and unstructuredness of a moduleâĂŹs control flow. A knot is a crossing of control structures, caused by an explicit jump out of a control structure either by break, continue, goto, or return. STKNT is undefined. Details and corresponding graphs can be found in D6.2 report.

Note: tables such as the one below contain values directly taken from the latest snapshots currently available on the QA-Verify web interface for Rephrase.

Table 2.1: Knot counts for Sequential and Parallel versions

Function	Sequential CPU version	Parallel CPU version
preprocessDSP	0	0
processCurves	0	0

From this table we can see that parallelisation did not introduce any difference on the knot counts.

2.1.2 Nesting and Cyclomatic Complexity

Description for these metrics can be found in D6.2 report.

2.1.2.1 Deepest Level of Nesting (STMIF)

Table 2.2: STMIF for Sequential and Parallel versions

Function	Sequential CPU version	Parallel CPU version
preprocessDSP	4	1
processCurves	3	3

Parallelisation removes 3 levels of nesting, as can be seen here: preprocessDSP sequential version has 4 level of nesting with 3 *for* loops and 1 *if* loop (Figure 2.3).

preprocessDSP Parallel version has only 1 *for* loop (Figure 2.4).

The situation is different for processCurves function where it shows 3 levels of nesting with 2 *for* loops and 1 *if* loop (Figure 2.5).

processCurves Parallel version also shows 3 levels of nesting, but only for the initialization code (Figure 2.6).

```
//filtering
for (unsigned int rcbIdx = 0; rcbIdx < numberOfRCBs; rcbIdx++) {

        float* inputSignal = RCBs[rcbIdx].rcbProc.rawData.data();
        float* filteredSignal = RCBs[rcbIdx].rcbProc.filteredData.data();
        unsigned int numOfSamples = RCBs[rcbIdx].rcbProc.rawData.size();

        for (unsigned int sampIdx = 0; sampIdx < numOfSamples; sampIdx++) {
                filteredSignal[sampIdx] = 0.0f;
                if (sampIdx >= numOfCoeffs - 1) {
                        float bp_temp = 0.0;
                        float hil_temp = 0.0;
                        for (unsigned int coeffIdx = 0; coeffIdx < numOfCoeffs; coeffIdx++) {
                                bp_temp += bp_coeffs[coeffIdx] * inputSignal[sampIdx - coeffIdx];
                                hil_temp += hil_coeffs[coeffIdx] * inputSignal[sampIdx - coeffIdx];
                        }
                        filteredSignal[sampIdx] = sqrt(bp_temp * bp_temp + hil_temp * hil_temp);
                }
        }

}
```

Figure 2.3: Sequential nesting preprocessDSP

```
//filtering

//Benchmark START
double startTime1= Utils::getTimeSec();
for (unsigned int rcbIdx = 0; rcbIdx < numberOfRCBs; rcbIdx++) {

        float* inputSignal = RCBs[rcbIdx].rcbProc.rawData.data();
        float* filteredSignal = RCBs[rcbIdx].rcbProc.filteredData.data();
        unsigned int numOfSamples = RCBs[rcbIdx].rcbProc.rawData.size();

        Task task {     inputSignal, bp_coeffs, hil_coeffs, filteredSignal, numOfSamples };
        StageInit stage_init(task);
        Stage stage { numOfCoeffs, numOfSamples };
        ff_Pipe<Task> pipe(stage_init, stage);

        pipe.run_and_wait_end();
}
```

Figure 2.4: Parallel nesting preprocessDSP

```
//iterate over each assigned measurement entity of every axle
for (axleIdx = 0; axleIdx < axleData.size(); axleIdx++) {
        for (measIdx = 0; measIdx < axleData[axleIdx].axleMeasEnts.size(); measIdx++) {
                if (axleData[axleIdx].axleMeasEnts[measIdx] != NULL) {

                        //choose the reference curve for the measurement entity
                        selectReference(axleIdx, measIdx);

                        //get converted curve of current entity
                        vector<int16_t> currSamples = axleData[axleIdx].axleMeasEnts[measIdx]->calcConvSamples();
                        vector<int> fittedCurve;

                        //perform curve fitting for current measurement entity
                        fitReference(*(axleData[axleIdx].axleMeasEnts[measIdx]), currSamples, fittedCurve);

                        //perform curve evaluation for current measurement entity
                        evalCurve(*(axleData[axleIdx].axleMeasEnts[measIdx]), currSamples, fittedCurve);
                }
        }

        //determine if measurement entities can be used for load calculation per-axle
        setIsUsable(axleData[axleIdx]);
}
```

Figure 2.5: Sequential nesting processCurves

```
// Collect valid axleIdx , measIdx pairs
vector <pair < unsigned , unsigned > > indices;
for (unsigned axleIdx = 0; axleIdx < axleData.size(); ++axleIdx) {
        vector <MeasEntity *>& axleMeasEnts = axleData[axleIdx].axleMeasEnts;
        for (unsigned measIdx = 0; measIdx < axleMeasEnts.size(); ++measIdx) {
                if (axleMeasEnts[measIdx] != 0) {
                        indices.push_back(std::make_pair(axleIdx, measIdx));
                }
        }
}

ParallelFor p;
p.parallel_for(0, indices.size () , [&indices, this](unsigned i) {

        unsigned axleIdx = indices[i].first;
        unsigned measIdx = indices[i].second;
        MeasEntity* currMeasEntity = axleData[axleIdx].axleMeasEnts[measIdx];

        //choose the reference curve for the measurement entity
        selectReference(axleIdx, measIdx);

        //get converted curve of current entity
        vector <int16_t> currSamples = currMeasEntity->calcConvSamples();
        vector <int> fittedCurve;

        //perform curve fitting for current measurement entity
        fitReference(*currMeasEntity, currSamples, fittedCurve);

        // perform curve evaluation for current measurement entity
        evalCurve(*currMeasEntity, currSamples, fittedCurve);
        });

for (unsigned axleIdx = 0; axleIdx < axleData.size(); ++axleIdx) {
        setIsUsable(axleData[axleIdx]);
}
```

Figure 2.6: Parllel nesting processCurves

It can therefore be said that the nesting has overall been reduced in the parallel version as it applies to less code.

2.1.2.2 Cyclomatic complexity (STCYC)

Description of STCYC can be found in D6.2 report.

Table 2.3: Cyclomatic complexity for Sequential and Parallel versions

Function	Sequential CPU version	Parallel CPU version
preprocessDSP	9	6
processCurves	4	5

Cyclomatic complexity for preprocessDSP function has reduced from 9 to 6 consistently with the 3 level of nesting reduction discussed above. The code outside the loop e.g. RCB decimation, JSON serialization, etc. is the same between the two versions. ProcessCurves function shows an increase by 1 of cyclomatic complexity due to the move of the setIsUsable call outside the main loop into its own loop, as can be seen in previous screenshot. This can be remedied by moving the initialization and finalization parts into separate functions (Table 2.3).

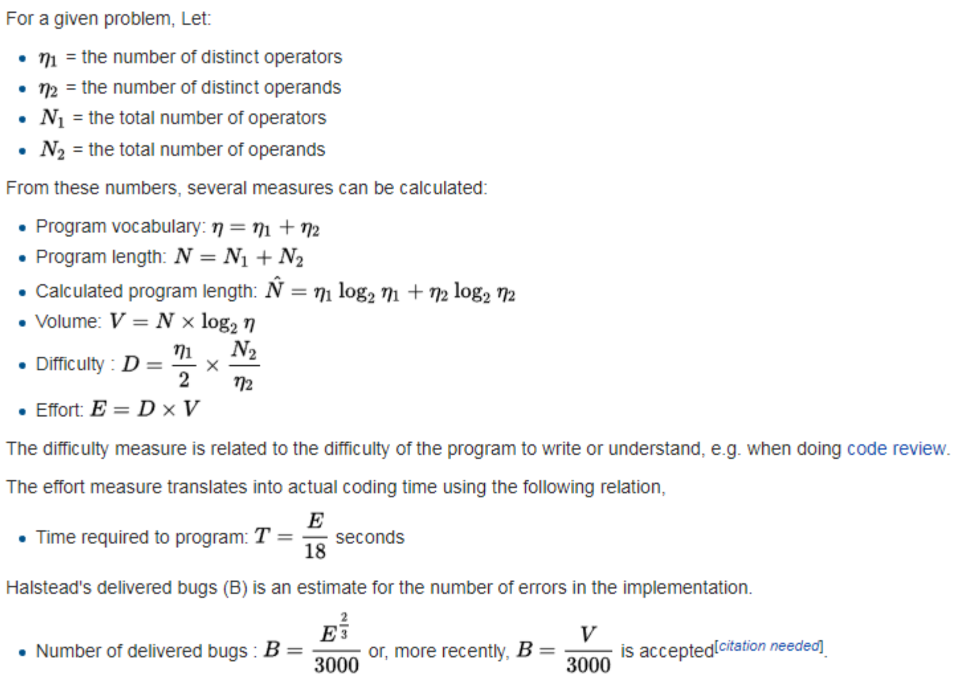
At project level, an increase of the Cyclomatic Complexity Across project (Reliability_STCYA, taken from STCYA to highlight contribution to Reliability) is observed (Table 2.4). This is due to refactoring between classes (mainProc.cpp, mainProc_transformed.cpp, MeasProcessor.cpp).

Table 2.4: Project level Cyclomatic complexity for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
Reliability_STCYA	1097	1104

2.1.3 Halstead metrics

STFDT: Number of distinct operators in a function; STFDN: Number of distinct operands in a function; STFN1: Number of operator occurrences in function; STFN2: Number of operand occurrences in function; Based on these metrics, we can compute the Halstead metric "Number of delivered bugs" (Figure 2.7). For more details see Halstead Metric.



For a given problem, Let:

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Calculated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

The difficulty measure is related to the difficulty of the program to write or understand, e.g. when doing code review.

The effort measure translates into actual coding time using the following relation,

- Time required to program: $T = \frac{E}{18}$ seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs : $B = \frac{E^{\frac{2}{3}}}{3000}$ or, more recently, $B = \frac{V}{3000}$ is accepted[citation needed].

Figure 2.7: Halstead metric

Results are shown in Table 2.5 (with B(n1,n2,N1,N2)).

Table 2.5: Halstead metric for Sequential and Parallel versions

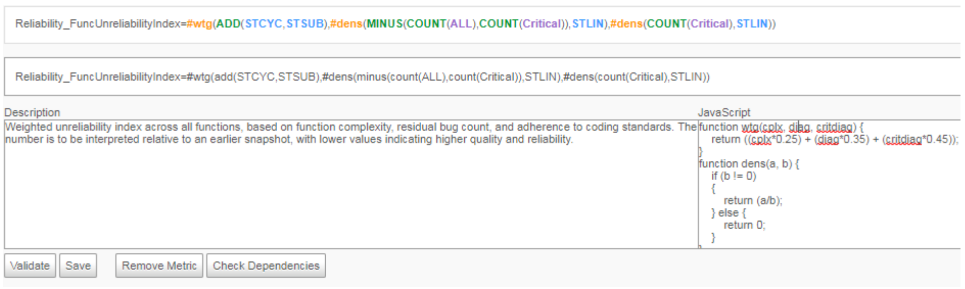
Function	Sequential CPU version	Parallel CPU version
preprocessDSP	B(26, 64,275,157) = 0.66	B(26,63,241,146) = 0.59
processCurves	B(20,17,97,51) = 0.27	B(23,32,149,80) = 0.37

The result appear consistent with the points mentioned ealier on the respective complexities of the functions.

2.1.4 Function Unreliability Index

Reliability_FuncUnreliabilityIndex is a compound metric, based on a quality formulae documented in QA-Verify Help pages (Figure 2.8). It is a weighted unreliability index across all functions based on function complexity, residual bug count, and adherence to coding standards. The number is to be interpreted relative to an earlier snapshot, with lower values indicating higher quality and reliability.

Complexity parameter is a function of cyclomatic complexity and program size (Cyclomatic complexity and Number of function calls). Residual bug count is the number of critical diagnostics identified by PRL analyzers, which is both 5 in CPU and CPU Parallel versions. As such the parallelisation did not introduce new critical issues. Adherence to coding standard is calculated as the total number of remaining diagnostics (non-critical), which is 307 and 304 in CPU and CPU Parallel respectively (Table 2.6).



```
Reliability_FuncUnreliabilityIndex=#wtg(ADD(STCYC,STSUB),#dens(MINUS(COUNT(ALL),COUNT(Critical)),STLIN),#dens(COUNT(Critical),STLIN))
```

```
Reliability_FuncUnreliabilityIndex=#wtg(add(STCYC,STSUB),#dens(minus(count(ALL),count(Critical)),STLIN),#dens(count(Critical),STLIN))
```

Description

Weighted unreliability index across all functions, based on function complexity, residual bug count, and adherence to coding standards. The number is to be interpreted relative to an earlier snapshot, with lower values indicating higher quality and reliability.

JavaScript

```
function wtg(cplx, diag, critdiag) {
   return ((cplx*0.25) + (diag*0.35) + (critdiag*0.45));
}
function dens(a, b) {
   if (b != 0)
   {
      return (a/b);
   } else {
      return 0;
   }
```

Figure 2.8: Function Unreliability Index Help

Table 2.6: Function Unreliability Index for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
Reliability_FuncUnreliabilityIndex	1306	1314

This appears consistent with the increase in Reliability_STCYA discussed earlier.

2.2 Integrity

2.2.1 Unreachable Code

Reliability_SumUNR is a compound metric computed as the project sum of function-based Number of unreachable statements (STUNR)and can be considered as the Integrity metric.

No unreachable statements was found in both sequential and parallel versions.

Additional integrity metrics can be uploaded to QA-Verify as described later in this document. A suggested metric is code coverage, based on a comprehensive test suite at the partnerâĂŹs site, which can be computed with a variety of free code coverage tools. OpenCppCoverage is very easy to set up and it is cross-platform.

2.3 Class-based Metrics

STCBO: Coupling to Other Classes (STCBO), Deepest Inheritance (STDIT), and Lack of cohesion within class (STLCM) are not directly applicable to the functions examined.

Project level aggregation (average) for these metrics can be computed in QA-Verify. Limiting the refactoring to the application of parallelisation patterns would help evaluation.

Chapter 3

Run Time Metrics

3.1 Time, Memory and Synchronisation

Timing is captured at function-level by surrounding the function call with the timing code already available in code:

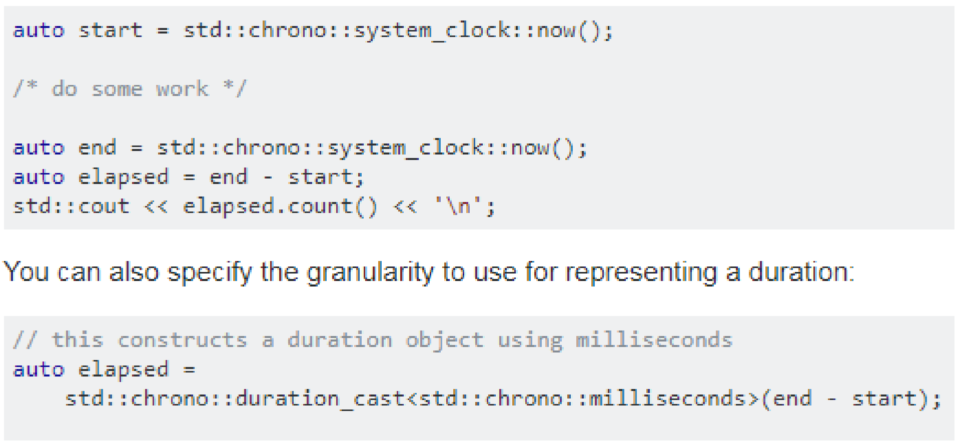
```
//perform curve fitting and curve evaluation steps for assigned measurement entities
double startTime2 = Utils::getTimeSec();
processCurves();
double endTime2 = Utils::getTimeSec();
std::cout <<"processCurves run time:" << endTime2 - startTime2 << "sec" << std::endl << std::endl;
```

Figure 3.1: Timing

```
double Utils::getTimeSec() {

        struct timespec ts;
        int timeStat;
        int currErrno;

        timeStat = clock_gettime(CLOCK_MONOTONIC, &ts);
        if (timeStat == -1) {
                currErrno = errno;
                throw OsExc(string("Error getting time: ").append(strerror(currErrno)));
        }

        return ts.tv_sec + ((double)ts.tv_nsec) / 1000000000.0;
}
```

Figure 3.2: Timing Implementation

C++11 users can use the std::chrono library:



```cpp
auto start = std::chrono::system_clock::now();

/* do some work */

auto end = std::chrono::system_clock::now();
auto elapsed = end - start;
std::cout << elapsed.count() << '\n';
```

You can also specify the granularity to use for representing a duration:

```cpp
// this constructs a duration object using milliseconds
auto elapsed =
    std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
```

Figure 3.3: Timing chrono

The result is then fed into metric Elapsed real (wall clock) time used by the function, in seconds (PFRTI)- Table 3.1.

Table 3.1: Speed comparison for Sequential and Parallel versions

Function	Sequential CPU version	Parallel CPU version
preprocessDSP	0.811	0.68
processCurves	0.105	0.043

This demonstrates speed up obtained from parallelisation. As per static analysis metrics, the data is captured for each snapshot into QA-Verify. Here is e.g. some measurements summarized in a Top10 chart for CPU Parallel - Figure 3.4.

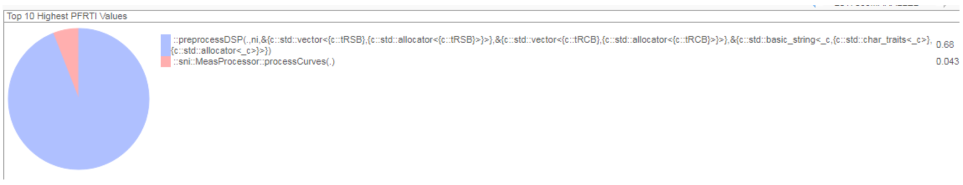
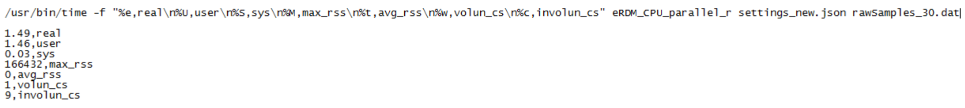


Figure 3.4: QA-Verify Speed Metrics

At program level, timing can be captured by surrounding main entry point, or better still using the time command (Linux) or equivalent (see Fugure 3.5).



```
/usr/bin/time -f "%e,real\n%U,user\n%S,sys\n%M,max_rss\n%t,avg_rss\n%w,volun_cs\n%c,involun_cs" eRDM_CPU_parallel_r settings_new.json rawSamples_30.dat

1.49,real
1.46,user
0.03,sys
166432,max_rss
0,avg_rss
1,volun_cs
9,involun_cs
```

Figure 3.5: Command Line Speed Metrics

See later section about how to get these values into QA-Verify. The real, user and sys fields are captured into 3 project-level metrics:

- Elapsed real (wall clock) time used by the process, in seconds (PPRTI)
- Total number of CPU-seconds used by the system on behalf of the process (PPSTI)
- Total number of CPU-seconds that the process used directly in user mode (PPUTI)

Results of these metrics are in Table 3.2.
PPUTI > PPRTI demonstrates speed up and parallelisation over multiple cores.

Table 3.2: Speed comparison for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
PPRTI	1.49	1.26
PPSTI	0.03	0.14
PPUTI	1.46	4.55

The other fields from *time* command output are captured into new metrics:

- Maximum resident set size of the process (PPMRS)
- Average resident set size of the process (PPARS)
- Number of times that the process was context-switched voluntarily (PPVCS)
- Number of times the process was context-switched involuntarily (PPICS)

Obtained results are in Table 3.3.

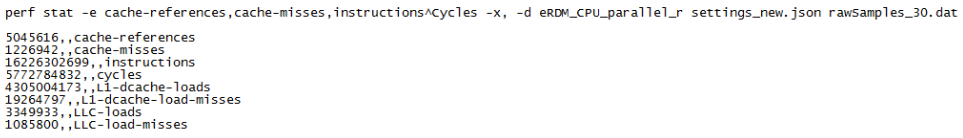
Table 3.3: Speed comparison for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
PPMRS	166432	166916
PPARS	0	0
PPVCS	1	11194
PPICS	9	476

PPMRS shows equivalent memory usage. An increased memory usage is usually an expected trade-off for higher throughput. A more extensive run and data set help compute better average memory usage statistics, as can be seen also with PPARS. PPVCS and PPICS demonstrate synchronisation taking place from parallelisation.

3.2 IPC and Cache Performance

Instructions per cycle (IPC) and cache performance can be captured on Linux via the *perf* utility (Figure 3.6).



```
perf stat -e cache-references,cache-misses,instructions^Cycles -x, -d eRDM_CPU_parallel_r settings_new.json rawSamples_30.dat

5045616,,cache-references
1226942,,cache-misses
16226302699,,instructions
5772784832,,cycles
4305004173,,L1-dcache-loads
19264797,,L1-dcache-load-misses
3349933,,LLC-loads
1085800,,LLC-load-misses
```

Figure 3.6: Perf Utility Metrics

The fields are captured into new project-level metrics:

- Number of cache references recorded for the process (PPCRF)
- Number of cache misses recorded for the process (PPCMS)
- Number of instructions recorded for the process (PPINS)

- Number of cycles recorded for the process (PPCYL)
- Number of L1 data cache loads recorded for the process (PL1DL)
- Number of L1 data cache misses recorded for the process (PL1DM)
- Number of last level data cache loads recorded for the process (PLCDL)
- Number of last level data cache misses recorded for the process (PLCDM)

Obtained results are in Table 3.4.

Table 3.4: Perf Utility Metrics for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
PPCRF	5045616	13589476
PPCMS	1226942	2447481
PPINS	16226302699	20278243928
PPCYL	5772784832	17274070834
PL1DL	4305004173	7224434499
PL1DM	19264797	28447717
PLCDL	3349933	7963606
PLCDM	19264797	28447717

These in turn feed the new compound metrics described thereafter. They are categorised as Adaptivity, as they are related to hardware resources.

3.3 Adaptivity

3.3.1 Instructions per Cycle

Number of instructions per cycle recorded for the process (Adaptivity_ProcessIPC) is a compound metric computed as PPINS / PPCYL.

In Table 3.5 we can see that the IPC actually decreased after parallelisation in this run, despite overall program speed up as seen earlier (PPRTI).

Table 3.5: Instructions per Cycle for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
Adaptivity_ProcessIPC	2.81	1.17

It is included here as it is a standard metric, but is sensitive to synchronisation timings during execution and is problematic as a before/after parallelisation metric - see e.g. IPC considered harmful for multiprocessor workloads

3.3.2 Cache Misses per Instruction

Ratio of cache misses to instructions recorded for the process (Adaptivity_ProcessMPI) is a compound metric computed as PPCMS / PPCRF. Obtained results are in Table 3.6.

Table 3.6: Cache Misses for Sequential and Parallel versions

Metric	Sequential CPU version	Parallel CPU version
Adaptivity_ProcessMPI	0.008	0.012

The ratio of cache-misses to instructions will give an indication on how well the cache is working; the lower the ratio the better (Ref.).Because of the relatively large difference in cost between the RAM memory and cache access (hundreds cycles vs. < 20 cycles) even small improvements of cache miss rate can significantly improve performance. If the cache miss rate per instruction is over 5%, further investigation is required.

Cache miss is slightly higher on the parallel version, but well under the suggested 5% threshold.

3.3.3 Cache Miss Rate

L1 data cache load miss rate recorded for the process (Adaptivity_ProcessL1CMR) is a compound metric computed as PL1DL/ PL1DM. Last level data cache load miss rate recorded for the process (Adaptivity_ProcessLLCMR) is a compound metric computed as PLCDL/ PLCDM. Cache miss rate recorded for the process (Adaptivity_ProcessCMR) is a compound metric computed as PPCMS / PPCRF.

Obtained results are in Table 3.7.

Table 3.7: Cache Miss Rate for Sequential and Parallel versions

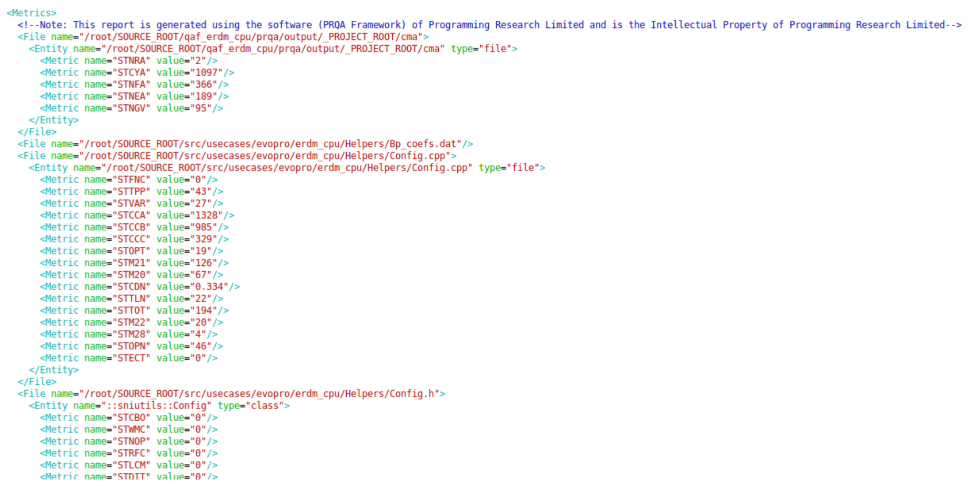
Metric	Sequential CPU version	Parallel CPU version
Adaptivity_ ProcessL1CMR	0.447	0.394
Adaptivity_ProcessLLCMR	32.413	26.673
Adaptivity_ProcessCMR	24.317	18.01

CPU Parallel in this run shows lower cache miss rates than the sequential version.

Chapter 4

Future Work

Due to confidentiality or set up requirements, PRL cannot run a binary for some of the use cases or access to the full test data set to profile the binary. While static analysis metrics produced by PRL provide valuable quality information, additional quality and performance metrics can be computed at the partners site and/or using Rephrase tools and fed into QA-Verify for integration into an existing QA-Verify snapshot. Currently, analysis by PRL tools of a use case code automatically publishes an archive of metric information to a shared FTP repository accessible by the partners. The metric report is an XML file - see Figure 4.1.



```
<Metrics>
  <!--Note: This report is generated using the software (PRQA Framework) of Programming Research Limited and is the Intellectual Property of Programming Research Limited-->
  <File name="/root/SOURCE_ROOT/qaf_erdm_cpu/prqa/output/_PROJECT_ROOT/cma">
    <Entity name="/root/SOURCE_ROOT/qaf_erdm_cpu/prqa/output/_PROJECT_ROOT/cma" type="file">
      <Metric name="STNRA" value="2"/>
      <Metric name="STCYA" value="1097"/>
      <Metric name="STNFA" value="366"/>
      <Metric name="STNEA" value="189"/>
      <Metric name="STNGV" value="95"/>
    </Entity>
  </File>
  <File name="/root/SOURCE_ROOT/src/usecases/evopro/erdm_cpu/Helpers/Bp_coefs.dat"/>
  <File name="/root/SOURCE_ROOT/src/usecases/evopro/erdm_cpu/Helpers/Config.cpp">
    <Entity name="/root/SOURCE_ROOT/src/usecases/evopro/erdm_cpu/Helpers/Config.cpp" type="file">
      <Metric name="STFNC" value="0"/>
      <Metric name="STTPP" value="43"/>
      <Metric name="STVAR" value="27"/>
      <Metric name="STCCA" value="1328"/>
      <Metric name="STCCB" value="985"/>
      <Metric name="STCCC" value="329"/>
      <Metric name="STOPT" value="19"/>
      <Metric name="STM21" value="126"/>
      <Metric name="STM20" value="67"/>
      <Metric name="STCDN" value="0.334"/>
      <Metric name="STTLN" value="22"/>
      <Metric name="STTOT" value="194"/>
      <Metric name="STM22" value="20"/>
      <Metric name="STM28" value="4"/>
      <Metric name="STOPN" value="46"/>
      <Metric name="STECT" value="0"/>
    </Entity>
  </File>
  <File name="/root/SOURCE_ROOT/src/usecases/evopro/erdm_cpu/Helpers/Config.h">
    <Entity name="::sniutils::Config" type="class">
      <Metric name="STCBO" value="0"/>
      <Metric name="STWMC" value="0"/>
      <Metric name="STNOP" value="0"/>
      <Metric name="STRFC" value="0"/>
      <Metric name="STLCM" value="0"/>
      <Metric name="STDIT" value="0"/>
```

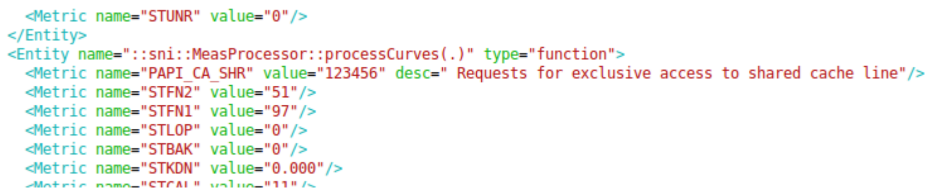
Figure 4.1: PRL Metrics Report

The sample above demonstrate 3 types of metrics:

- A project-level metric is listed under entity name ending with _PROJECT_ROOT_/cma.
- A file-level metric is listed under entity name containing the full path to the file.

- A class-level metric is listed under entity name containing the qualified name to the class/function e.g. ::sniutils::Config, which represents the *Config* class in namespace *sniutils*. The *type* attribute is set to *class*. Likewise function-level metric can be specified with *type* set to *function*.

All files, classes and functions in the project are listed in this file, because there is always at least one associated metric produced by PRL analysers. To specify a description for a new metric, which serves as documentation in QA-Verify, an optional attribute *desc* is used. For example, using the PAPI tool from D3.2, a new function-level metric Requests for exclusive access to shared cache line (PAPI_CA_SHR) is captured for the processCurves function (See Figure 4.2).



```
  <Metric name="STUNR" value="0"/>
</Entity>
<Entity name="::sni::MeasProcessor::processCurves(.)" type="function">
  <Metric name="PAPI_CA_SHR" value="123456" desc=" Requests for exclusive access to shared cache line"/>
  <Metric name="STFN2" value="51"/>
  <Metric name="STFN1" value="97"/>
  <Metric name="STLOP" value="0"/>
  <Metric name="STBAK" value="0"/>
  <Metric name="STKDN" value="0.000"/>
  <Metric name="STCAL" value="11"/>
```

Figure 4.2: New Metric

The modified XML should be renamed after the target QA-Verify snapshot and pushed onto *metric_feed* subdirectory at the base of the use case directory in the Rephrase Git repository, e.g. erdm_cpu/metric_feed/20170630162516_CPU_PARALLEL.xml.

The new metric will automatically be integrated into the target QA-Verify snapshot, and can be subsequently analysed in QA-Verify as per the other metrics (Metric Trend graph, Top10, etc.).

Chapter 5

Conclusion

Analysis of the static analysis and runtime metrics produced by PRL tools demonstrate on the Railway sample that the application of Rephrase parallelisation patterns maintain and sometimes enhance the target quality metrics , and enhance performance. Other use cases will likewise be analysed and interpreted in order to provide a fuller picture. A framework to introduce new metrics has been presented, giving the opportunity to leverage PRL tools to capture and analyse new data produced by the partners in their testing environment, using their tools or D3.2 Rephrase tools, thus giving the opportunity to enhance the evaluation of reliability, robustness, resilience, integrity, and adaptivity for the final D6.7 report.