Project no. 644235

REPHRASE

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

Selection of Evaluation Metrics
D6.2

Due date of deliverable: 30.06.2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP6

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

Version 0.3

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	30/6/16	Evgueni Kolossov	PRL	Original Version
2	11/10/16	Evgueni Kolossov	PRL	Minor improvements

Executive Summary

The deliverable is the description of the approach for selection and implementation of evaluation metrics for assessment of the technology and porting success. for evaluation in T6.3 different metrics will be developed to measure reliability, robustness, resilience, integrity, and adaptivity before and after initial and final porting.

Table 1: Definitions of software reliability, robustness, resilience, integrity and adaptivity, and associated metrics

	Definition	**Associated Metric**
Reliability	The probability of failure-free operation over a specified time in a given environment for a specific purpose	MTTF (Mean Time To Failure) on specific hardware and with predicted input values
Robustness	A program that performs well not only under ordinary conditions, but also under special conditions that stress its designers' assumptions	MTBF (Mean Time Between Failures) on specific hardware with possibly unpredicted input values
Resilience	The ability for a system to take a hit to a critical component and recover and come back for more in a known, bounded, and generally acceptable period of time	Validation correct and continued operation under injected problems, based on modelling the kinds of problems the system is expected to endure
Integrity	The measure of quality of the source code in terms of how thoroughly the code is tested to reach high code coverage	Using the ratio between covered and total code elements
Adaptivity	The ability for the program to adapt to changes in the environment, such as the availability of hardware resources	MTBF when executed on different hardware platforms and with different amounts of resources (such as memory and available cores

Contents

Chapter 1

Introduction

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.

For example, the following metrics cannot be measured by automated static analysis tools (ASA):

- 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 modeling 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.

To get valuable results from these metrics, it would be probably necessary to manually compute them in each software or to cope with a simulation approach. For the MTTF or MTBF metric computations, simulations based on Markov's models could be employed. The resilience computation would probably involve application of some automatic test driver, which would run the given application and simulate the expected problems. For the integrity computation, C++ code coverage tool (dynamic) shall be used.
As can be seen, ASA tools are not suitable for the measurement of the metrics in discussion. Therefore, it can be beneficial to search for other metrics, which could cover the specified software characteristics, and at the same, to be more suitable for measurement via ASA tools, especially via tools developed by *Programming Research Limited (PRL)*. Metrics more suitable for measurement are also mentioned in T6.1 (cyclomatic complexity, program flow analysis measures ...).
To achieve this goal, we need to analyse the reliable approaches for metrics selection. This will allow to calculate using existing software or utilities and which will be useful not only for particular test case but for any kind of test cases.

Chapter 2

ISO 25000

2.1 ISO 25000 Metrics

Software metrics and their measurement in the context of software quality is well covered in ISO/IEC norm 25000 known as SQuaRE [3], which is a successor of ISO/IEC 9126 [1]. The ISO 25000 covers the problematic of the quality evaluation of software systems from different aspects (internal quality, external quality, etc.). The main assets of the norms are a guide for software quality evaluation together with a computational framework for software measures and list of suggested metrics for software evaluation.

The general benefits from usage of the norm are:

- The norm describes and provides guidance of widely accepted and standardized approach of how to measure software (described in Section 2.1.1).
- The norm allows the definition of new "base metrics" (called Quality Measure Elements, QMEs) together with definition of new "compound metric" (called Quality Measures, QMs) and provide guidance how to work with them.
- The norm provides stable terms and vocabulary from the area of software measures.

The benefits that we could gather from it's usage are the following:

- The norm specifies a computational model which can be used as a reference when providing explanation about how we measure selected software characteristic (robustness, resilience, etc.). Due to we will not be using the norm completely, we won't be able to say: we do it as it is described in the norm. But we could say something in the sense: we used some of the ideas from ISO/IEC 25000'.

- The norm contains a suggestion list of compound metrics (approximately 70, in the context of the norm are called Quality Measures) together with their definitions. Most of the metrics are not suitable to be measured by ASA tools (see Appendix A for a complete list). Nevertheless, we can refer to the norm for example when evaluation reliability (see bold in Appendix A).

- We could define another metrics for selected characteristics in internationally accepted format.

- The resilience computation is defined in detail in one part of the norm (ISO/IEC 25045) and can be used as a guidance if we would stay with the current definition of the resilience metric (the current metric definition and the metric definition in the norm are almost identical).

- The norm suggests "base metrics" that we could possible to reuse (see both Appendix B and Appendix C).

- There could be a possible output in a form of a scientific article (topic: "The selection of quality model for evaluation of parallel applications").

What the norm does not say and what is still going to be our responsibility (even if we were using some bits and pieces from the norm) is the process of finding/defining the right metrics for our selected characteristics. The work dedicated to find adequate metrics that can be lately relatively easy to measure by PRL tools would still stay the same (see Subsection 'Suggested steps').

Nevertheless, by using the computational framework from the norm, we could define most of the metrics in order to measure them more easily by ASA tools. An example can be the reliability characteristic. One of the possible base measurements when using the ASA tool from PRL can be the number of high level diagnostics found in the source code. The metric with name $HLOC$ can be defined as number of high level diagnostics/LOC and can be used as one of the metrics for evaluating the reliability.

We could also use the "compounding" by defining metrics like 'MC++' as the number of high level diagnostics from MISRA C++ found in the source code/LOC (code compliance standard checking beside other things reliability problems). Afterwards, we can define JSF as the number of high level diagnostics found in the source code /LOC and combine them together. Weights can be used when combining the metrics. There are almost no limits on the number of metrics one can define. The main restriction is to provide sufficient justification for each metric. Each base metric and compound metric must contain a clear description why it could be used for measurement of given characteristics.

2.1.1 SQuaRE (Software Product Quality Requirements and Evaluation)

SQuaRE provides a framework for evaluating the quality of software products and states the requirements for methods of software product measurement and evaluation. SQuaRe can define: "The SQuaRE describes what to evaluate but does not specify how. In other words, it does not detail the thresholds for the evaluation metrics to be used, nor it describes how to grop these metrics in order to assign a quality value to a software product." as described in [2].

The concept of the SQuaRE is shown in Figure 2.1. In simple terms:

- The quality is divided into "quality categories" called $characteristics$.
- The characteristics are divided into "quality subcategories" called sub-characteristic.
- Each sub-characteristic or characteristic has list of Quality Measures: compound metrics used for measurement.
- Each Quality Measure is compound of Quality Measure Elements ("base metric").

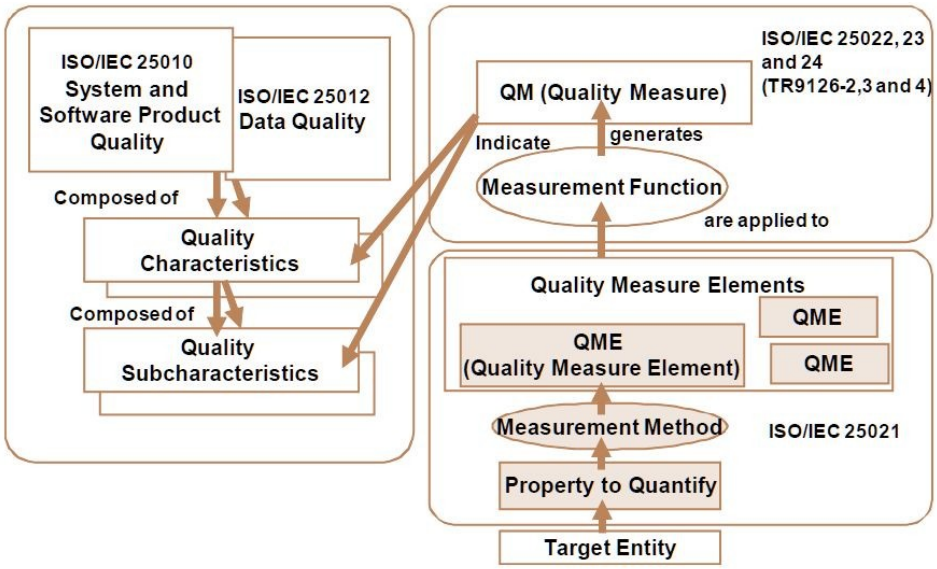


Figure 2.1: The general model structure in SQuaRE.

The application of SQuaRE is shown in Figure 2.2. In simple terms:

- Failures and duration are gathered from the system by a measurement method creating two QMEs: $Numberof failures$ and $Durationperiod$.
- These two QMEs are used in QM (called MTBF), which is computed as $Numberof failures/Durationperiod$.
- This Quality Measure indicates the assigned quality $characteristic/sub-characteristic$.

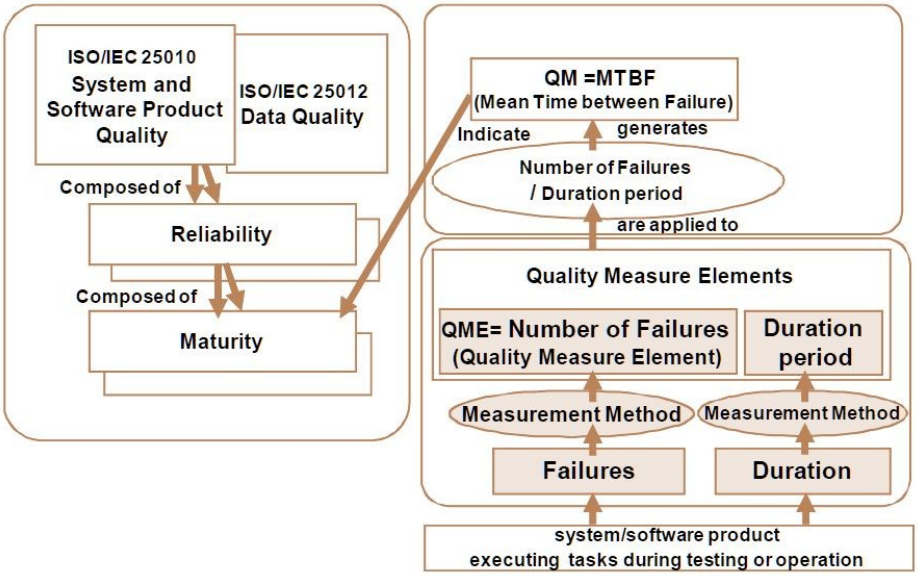


Figure 2.2: Example of usage of the model.

The SQuaRE describes a general software quality measurement model, which can be applied across all software types for software quality evaluation. The general quality model is not well suited for us, given that it covers quality in general (contains multiple views, many other software characteristics etc.) while our intent is to evaluate several selected software characteristics in order to demonstrate the benefits of software produced by RePhrase methodology against typical existing software.

2.1.2 Using SQuaRE for Rephrase

Selected software characteristics (Reliability, Robustness, Adaptivity, Resilience, and Integrity) are hard to measure directly by ASA tools. Therefore, if there is no direct measure for the selected software characteristic, we should find out software entities, which can be measured in order to evaluate the characteristic (QMEs). If suitable, instead of measuring one property (QME), we can identify collection of

properties (QMEs) that cover selected characteristic, measuring them and combine them computationally to arrive at a derived Quality Measure which corresponds to given characteristic.

This procedure can be applied on more levels. We can divide the characteristics into characteristics and find measurable properties of the sub-characteristics. The number of levels is not limited and it should be chosen to provide the best fit for the chosen characteristics.

Chapter 3

CIBERSAM quality metrics

In order to compare the image quality of the use cases provided by CIBERSAM, we rely this measurement on the Signal-to-Noise ratio (SNR) metric [6]. SNR is used in the Huygens Software as a Regularization Parameter, i.e. as a parameter that controls the sharpness of the restoration result. The higher this value, the sharper your restored image will be. The correct method to estimate the SNR consist of finding the Standard Deviation (STD) around a maximum level.

Listing 3.1 summarizes the SNR calculation in pHARDI use case. This process is calculated for each volume in the input data. After a determined number if iterations, it returns the average calculated SNR value (line 20). As explained before, the SNR is based in the computation of the STD related to the maximum value of the data (line 14).

Listing 3.1: Calculation of SNR in the pHARDI use case.

```
1
2  for (size_t i = 0; i < Niter; ++i) {
3      .....
4
5      for (size_t k = 0; k < Signal.n_cols; ++k ) {
6          for (size_t j = 0; j < Signal.n_rows; ++j) {
7              SUM(j,k) = (pow(Signal(j,k),2) + pow(Reblurred(j,k),2))/2 - (
                   (cont.)sigma2(j,k) * Reblurred_S(j,k)) * Ratio(j,k) ;
8          }
9      }
10
11     sigma2_i = (1.0/N) * sum( SUM , 0) / n_order;
12
13     for (size_t k = 0; k < sigma2_i.n_elem; ++k ) {
14         sigma2_i(k) = std::min<T>(std::pow<T>(1.0/10.0,2),std::max<T>(
               (cont.)sigma2_i(k), std::pow<T>(1.0/50.0,2)));
15     }
16
17     sigma2 = repmat(sigma2_i, N, 1);
18 }
19
20 mean_SNR = mean (1.0 / sqrt(sigma2_i));
```

Currently, this metric is integrated in the provided use case. In a near future,

we plant to extract this functionality from the source code, facilitating its usage and integration with other existing tools (i.e. QA-Verify).

Examples of calculated SNR are shown in Figure 3.1. The top figure represents the mean squared error (MSE) compared with the obtained SNR. These metrics are commonly used in image processing frameworks.

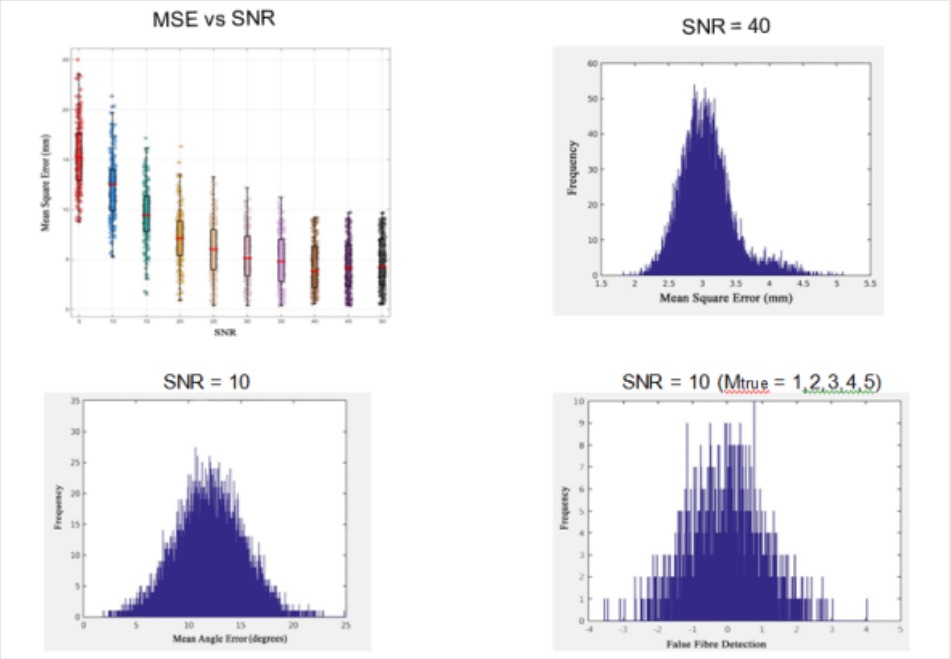


Figure 3.1: Examples of SNR distributions.

Chapter 4

Using TYPEPERF command

In PRL, we have developed a tool that generates a $TYPEPERF$ command for any process name and any specified maximum number of instances passed as arguments to the script then aggregates them through AWK to produce a CSV output (with a header for the counters). A graph can then be computed over time, along with some essential statistics such as min, max, average, and deviation.

The counters were chosen based on the MSDN recommendation for memory leaks. We can adjust these to cope with parallel performance profiling by selecting additional counters. For example, this document suggests the following performance counters to monitor performance (ignoring the SQL Server ones[1]).

Some metrics could map directly onto the output of some counters, like these ones which provides the per-CPU processor activity: Processor and % Processor Time (the primary indicator of processor activity).

Others could be derived, e.g. the thread queue size divided by duration gives us average queue size, thus giving an indication of performance on parallelism within a process: System->Processor Queue Length, Threads waiting to execute (<3).

As a generalization, we could provide statistics computed on the CSV output data and possibly later import these data into QA-Verify application. The outputs contain the following information [4]:

- Memory - Available Bytes reports available bytes; its value tends to fall during a memory leak.
- Memory - Committed Bytes reports the private bytes committed to processes; its value tends to rise during a memory leak.
- Process (process_name) - $PrivateBytes$ reports bytes allocated exclusively for a specific process; its value tends to rise for a leaking process.
- Process (process_name) - $WorkingSet$ reports the shared and private bytes allocated to a process; its value tends to rise for a leaking process.

[1] http://ss64.com/nt/syntax-performance-counters.html

- Process (process_name) - $PageFaults/sec$ reports the total number of faults (hard and soft faults) caused by a process; its value tends to rise for a leaking process.
- Process (process_name) - $PageFileBytes$ reports the size of the paging file; its value tends to rise during a memory leak.
- Process (process_name) - $HandleCount$ reports the number of handles that an application opened for objects it creates. Handles are used by programs to identify resources they must access. The value of this counter tends to rise during a memory leak; however, you cannot rule out a leak simply because this counter's value is stable.

This information can be used to compare projects before and after parallelization and probably can in future insert these values as a custom metrics into QA-Verify.

Chapter 5

Proposed complexity metrics for parallelisation

5.1 Control flow metrics

5.1.1 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 module's 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 for functions with unreachable code. Figure 5.1 explains the Knots metrics.

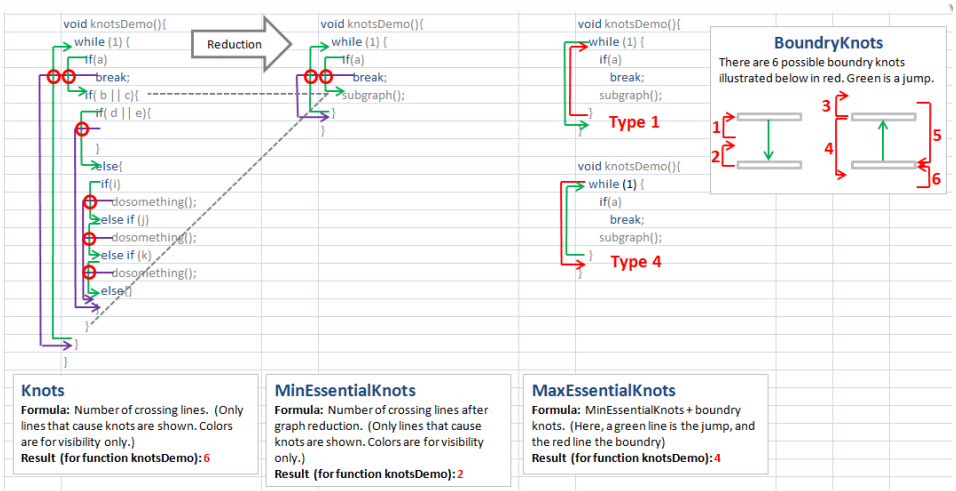


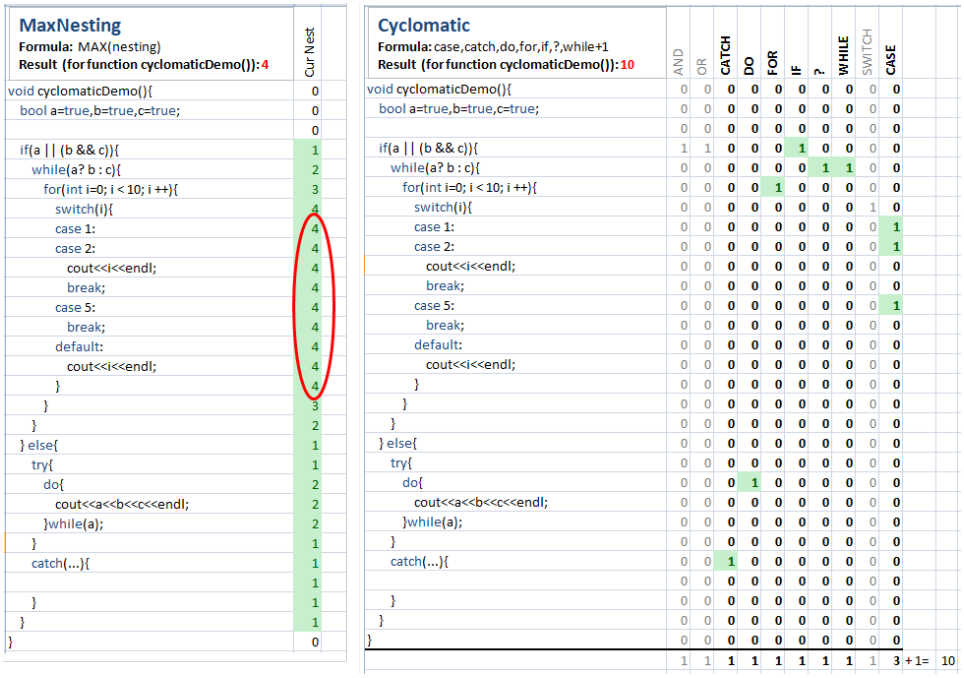
Figure 5.1: Knots metrics [5].

5.1.2 Nesting and cyclomatic complexity

Deepest level of nesting metric is implemented inside PRL software (A.2.21 STMIF: Deepest Level of Nesting). This metric is a measure of the maximum control flow nesting in your source code. You can reduce the value of this metric by turning your nesting into separate functions. This will improve the readability of the code by reducing both the nesting and the average cyclomatic complexity per function.

Cyclomatic Complexity metric is implemented inside PRL software (A.2.5 STCYC: Cyclomatic Complexity). Cyclomatic complexity is calculated as the number of decisions plus 1. High cyclomatic complexity indicates inadequate modularization or too much logic in one function. Software metric research has indicated that functions with a cyclomatic complexity greater than 10 tend to have problems related to their complexity.

Figure 5.2 depicts the Nesting and cyclomatic complexity metrics.



MaxNesting
Formula: MAX(nesting)
Result (for function cyclomaticDemo()): 4

	Cur Nest
void cyclomaticDemo(){	0
bool a=true,b=true,c=true;	0
	0
if(a || (b && c)){	1
while(a? b : c){	2
for(int i=0; i < 10; i ++){	3
switch(i){	4
case 1:	4
case 2:	4
cout<<i<<endl;	4
break;	4
case 5:	4
break;	4
default:	4
cout<<i<<endl;	4
}	4
}	3
}	2
} else{	1
try{	1
do{	2
cout<<a<<b<<c<<endl;	2
}while(a);	2
}	1
catch(...){	1
	1
}	1
}	1
}	0

Cyclomatic
Formula: case,catch,do,for,if,?,while+1
Result (for function cyclomaticDemo()): 10

	AND	OR	CATCH	DO	FOR	IF	?	WHILE	SWITCH	CASE	
void cyclomaticDemo(){	0	0	0	0	0	0	0	0	0	0	
bool a=true,b=true,c=true;	0	0	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	0	0	
if(a || (b && c)){	1	1	0	0	0	1	0	0	0	0	
while(a? b : c){	0	0	0	0	0	0	1	1	0	0	
for(int i=0; i < 10; i ++){	0	0	0	0	1	0	0	0	0	0	
switch(i){	0	0	0	0	0	0	0	0	1	0	
case 1:	0	0	0	0	0	0	0	0	0	1	
case 2:	0	0	0	0	0	0	0	0	0	1	
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0	
break;	0	0	0	0	0	0	0	0	0	0	
case 5:	0	0	0	0	0	0	0	0	0	1	
break;	0	0	0	0	0	0	0	0	0	0	
default:	0	0	0	0	0	0	0	0	0	0	
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
} else{	0	0	0	0	0	0	0	0	0	0	
try{	0	0	0	0	0	0	0	0	0	0	
do{	0	0	0	1	0	0	0	0	0	0	
cout<<a<<b<<c<<endl;	0	0	0	0	0	0	0	0	0	0	
}while(a);	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
catch(...){	0	0	1	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
}	0	0	0	0	0	0	0	0	0	0	
	1	1	1	1	1	1	1	1	1	3	+ 1 = 10

Figure 5.2: Nesting and cyclomatic complexity metrics.

5.1.3 Halstead metrics

This is metrics where operators and operands are counted only if they are directly used by the patterns of the paralellisation frameworks such as Fastflow, OpenCL etc.

The number of Distinct Operands metric is implemented inside PRL software (A.3.18 STOPN: Number of Distinct Operands). This is the number of distinct

operands used in the file. Distinct operands are defined as unique identifiers and each occurrence of a literal. Most literals, except 0 and 1, are usually distinct within a program. Since macros are usually used for fixed success and failure values (such as TRUE and FALSE), the differences in counting strategies are fairly minimal.

Number of Distinct Operators metric is implemented inside PRL software (A.3.19 STOPT: Number of Distinct Operators). This covers any source code tokens not supplied by the user, such as keywords, operators, and punctuation. STOPT is used in the calculation of a number of other metrics.

5.2 Object-oriented metrics

5.2.1 Coupling to other classes

Coupling to Other Classes metric is implemented inside PRL software (A.4.1 STCBO: Coupling to Other Classes). This is a count of the number of methods (member functions) or member objects of other classes accessed by a class. Only classes outside the inheritance hierarchy are considered because you would expect interaction with base and derived classes. Coupling to classes outside the inheritance tree should be viewed with suspicion as it makes the class less independent and less re-usable. This is one of Chidamber and Kemerer's suite of object oriented metrics. Figure 5.3 explains the Coupling to other classes metrics.

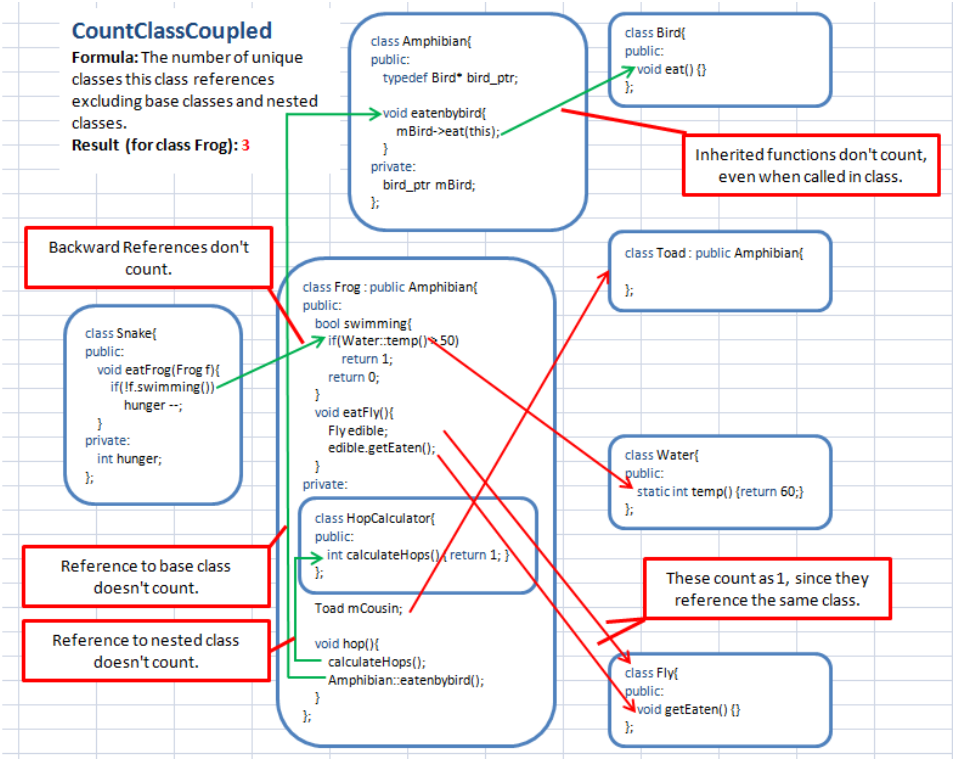


Figure 5.3: Coupling to other classes metrics.

5.2.2 Deepest inheritance

Deepest Inheritance metric is implemented inside PRL software (A.4.2 STDIT: Deepest Inheritance). This represents the number of derivations from the furthest base class down to this class. A high figure may indicate that the class depends on accumulated functionality, which makes understanding the class potentially difficult. This is one of the metrics defined by Chidamber and Kemerer. Figure 5.4 shows the Deepest inheritance metrics.

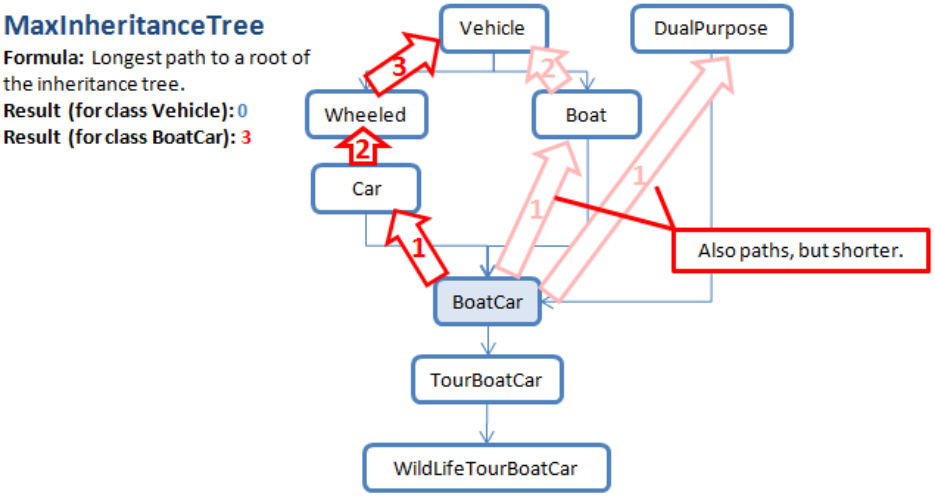


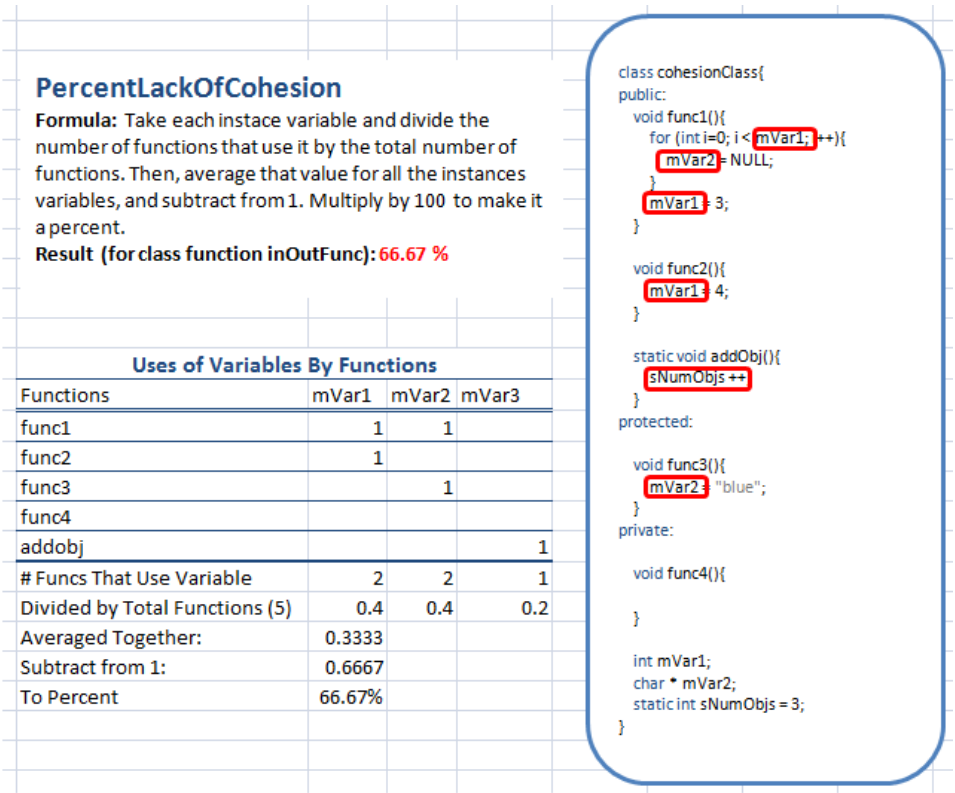
Figure 5.4: Deepest inheritance metrics.

5.2.3 Deepest template

Deepest template metric represent the longest path to a root of the template declaration tree.

5.2.4 Lack of cohesion within class

Lack of cohesion within class metric (STLCM) is implemented inside PRL software (A.4.3 STLCM: Lack of Cohesion within Class). Methods within a class are partitioned into sets that access independent sets of member objects. The STLCM metric is a count of sets of unrelated members for a type. Figure 5.5 explains the lack of cohesion within class metrics.



PercentLackOfCohesion

Formula: Take each instace variable and divide the number of functions that use it by the total number of functions. Then, average that value for all the instances variables, and subtract from 1. Multiply by 100 to make it a percent.

Result (for class function inOutFunc): 66.67 %

Uses of Variables By Functions

Functions	mVar1	mVar2	mVar3
func1	1	1	
func2	1		
func3		1	
func4			
addobj			1
# Funcs That Use Variable	2	2	1
Divided by Total Functions (5)	0.4	0.4	0.2
Averaged Together:	0.3333		
Subtract from 1:	0.6667		
To Percent	66.67%		

```
class cohesionClass{
public:
  void func1(){
    for (int i=0; i < mVar1; ++){
      mVar2 = NULL;
    }
    mVar1 = 3;
  }

  void func2(){
    mVar1 = 4;
  }

  static void addObj(){
    sNumObjs ++
  }
protected:

  void func3(){
    mVar2 = "blue";
  }
private:

  void func4(){

  }

  int mVar1;
  char * mVar2;
  static int sNumObjs = 3;
}
```

Figure 5.5: Lack of cohesion within class.

Chapter 6

Future work

The future work will consists of the following steps:

- Finalise the list of metric we want to have to measure the difference before and after parallelization.
- Gathering base metrics (Quality Measure Elements): The target of this step is to review and collect all base metrics (QMEs) we are able to measure at the moment by our tools and from the integrated already tools which correspond or can contribute into first item on the list.
- Investigate the possibility of usage of the generates diagnostics as base metrics i.e. "number of high level diagnostics" produced by PRL software.
- Investigate possibility creation of compound metrics (Quality Measures) from the base metrics (QME): The target of this step is to prepare list of compound metrics (QMs) which would cover the selected characteristics (robustness, resilience, etc. and which would be possible to use in QA-Verify.
- Investigate using survey for software vendors the possible third-party tools which can provide missing metrics for reliability, robustness, resilience, integrity and adaptivity, and investigate possibility of integration into QA-Verify.
- If above-mentioned steps did not bring coverage of all characteristics, investigate the possibility of manual measurement of given characteristics by usage of ISO 25000 (see ISO/IEC 25045 for semi-automatic approach of resiliency measurement).

Chapter 7

Conclusion

Analysis of requirements and availabilities of the metrics for measure of the reliability, robustness, resilience, integrity, and adaptivity indicates that most of the metrics cannot be calculated using static analysis and may require an additional integration with dynamic analysis tools or special utilities.

Metrics which available via static analysis are available in PRL QA-Verify software and will be used for analysis of the RePhrase use case projects. An additional work required to investigate useful metrics that can be combined from the existing metrics to provide a single measure for the success of the parallelisation.

Some additional work required for implementation of different methods for metrics calculations that described in this report.

Appendices

Appendix A

list of compound metrics (Quality Measures) from ISO/IEC 91263

The successors of ISO/IEC 91263, the ISO/IEC 25023 has not been published yet and it is recommended by ISO/IEC 25021 to use the previous version until it is published.

The list is in the form:

Characteristic

- Sub-characteristic / Metric (Quality Measure) in case no other level
- Metric (Quality Measure)

Suitability

- Functional adequacy
- Functional implementation completeness
- Functional implementation coverage
- Functional specification stability (volatility)

Accuracy

- Computational Accuracy
- Precision

Interoperability

- Data exchangeability (Data format based)
- Interface consistency (protocol)

Security

- Access auditability
- Access controllability
- Data corruption prevention
- Data encryption

Functionality

- Functional compliance
- Intersystem standard compliance

Reliability

- Maturity
  - Fault detection
  - Fault removal
  - Test adequacy
- Fault tolerant metrics
  - Failure avoidance
  - Incorrect operation avoidance
- Recoverability
  - Restorability
  - Restoration Effectiveness
- Reliability compliance
  - **Reliability compliance (How compliant is the reliability of the product to applicable regulations, standards and conventions?)**

Usability

- Understandability
  - Completeness of description
  - Demonstration capability
  - Evident functions
  - Function understandability
- Learnability

  - Completeness of user documentation and/or help facility

- Operability
  - Input validity checking
  - User operation cancellability
  - User operation Undoability
  - Customisability
  - Physical accessibility
  - Operation status monitoring capability
  - Operational consistency
  - Message Clarity
  - Interface element clarity
  - Operational error recoverability

- Attractiveness
  - Attractive interaction
  - User Interface appearance customisability

- Usability compliance
  - Usability compliance

Efficiency

- Time behaviour
  - Response time
  - Throughput time
  - Turnaround time

- Resource utilisation
  - I/O Utilization
  - I/O Utilization Message Density
  - Memory utilization
  - Memory utilization message density
  - Transmission Utilization

- Efficiency compliance
  - Efficiency Compliance

Maintainability

- Analysability
  - Activity recording
  - Readiness of diagnostic function
- Changeability
  - Change recordability
- Stability
  - Change impact
  - Modification impact localization
- Testability
  - Completeness of built-in test function
  - Autonomy of testability
  - Test progress observability
- Maintainability compliance
  - Maintainability compliance

Portability

- Adaptability
  - Adaptability of data structures
  - Hardware environmental adaptability
  - Organisational environment adaptability
  - Porting user friendliness
  - System software environmental adaptability
- Installability
  - Ease of Setup Retry
  - Installation effort
  - Installation flexibility
  - Available coexistence
- Replaceability
  - Continued use of data
  - Function inclusiveness
- Portability compliance
  - Portability compliance

Appendix B

List of metrics (Quality Measure Elements) from ISO/IEC 91263

The successors of ISO/IEC 91263, the ISO/IEC 25023 has not been published yet and it is recommended by ISO/IEC 25021 to use the previous version until it is published.

This list is part of "Pure Internal metrics" list, which are defined int 91263 as: "Pure Internal metrics are used to measure certain attributes of the software design and code of the software product that will influence the same or all of the overall software characteristics and sub-characteristics". They are equal to 25000 QMEs

Coherence

- Traceability
- Cyclomatic number
- Information Flow Complexity

Self-descriptiveness

- Modularity

Self-containedness

- Program size
- Conditional statement
- Unified data reference
- Adequacy of variable names
- Data-coupled module ratio
- Program statements

- Average module size
- Function coupled module ratio

Appendix C

List of metrics (Quality Measure Elements) from ISO/IEC 25021

The metrics are as follows:

- Number of accessible functions Number of user problems Number of records
- Duration
- Effort (in unit of time) Number of system failures Number of failures
- Number of faults
- Functional size of a product Number of data items Number of error messages Number of errors
- Number of messages
- Number of steps(in procedure) Task complexity
- Number of test cases Number of use cases Number of operations Number of fatal errors Size of a database Size of a memor
- QMEs extensions (?)
- Duration of Processing (time between start point to and finish point of specified task of system or software)
- Duration of Service Available Duration of Response Duration of Operator Work Duration of Operation
- Duration of Restoration Duration of System Migration Duration of Modification Number of Users
- Number of Users (Number of authorized users) Number of Interfaces

- Size of logs(Number of logs) Number of Document
- Number of System Handling Equipments and Site Locations
- Number of Connections
- Number of resources (including assets) Number of components
- Data volume
- Number of requirements
- Throughput
- Number of Tolerable Faults
- Number of Change Requests on User Interface

Bibliography

[1] ISO/IEC 9126 Software engineering – Product quality was an international standard for the evaluation of software quality. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=35786, 2014.

[2] Software Product Quality Evaluation Using ISO/IEC 25000. http://ercim-news.ercim.eu/en99/special/software-product-quality-evaluation-using-iso-iec-25000, 2014.

[3] Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64764, 2014.

[4] Investigating User-Mode Memory Leaks. https://technet.microsoft.com/en-us/library/cc938582.aspx, Accessed 2016.

[5] What Metrics does Understand have? https://scitools.com/support/metrics_list/, Accessed 2016.

[6] Erick Jorge Canales-Rodríguez, Yasser Iturria-Medina, Yasser Alemán-Gómez, Lester Melie-García, E J Canales-Rodriguez, Y Aleman-Gomez, and L Melie-Garcia. Deconvolution in diffusion spectrum imaging. *NeuroImage*, 50(1):136–149, 2010.