Project Number 688146

D3.2 – Final report on programmer- and productivity-oriented software tools

Version 1.0
4 September 2018
Final

Public Distribution

Easy Global Market, Wings ICT Solutions, HLRS, Unparallel Innovation

Project Partners: **Easy Global Market, GMV, Intecs, The Open Group, University of Stuttgart, University of York, Unparallel Innovation, WINGS ICT Solutions**

PROJECT PARTNER CONTACT INFORMATION

Easy Global Market Philippe Cousin 2000 Route des Lucioles Les Algorithmes Batiment A 06901 Sophia Antipolis France Tel: +33 6804 79513 E-mail: philippe.cousin@eglobalmark.com	**GMV** José Neves Av. D. João II, Nº 43 Torre Fernão de Magalhães, 7º 1998 - 025 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Intecs Silvia Mazzini Via Umberto Forti 5 Loc. Montacchiello 56121 Pisa Italy Tel: +39 050 9657 513 E-mail: silvia.mazzini@intecs.it	**The Open Group** Scott Hansen Rond Point Schuman 6 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Stuttgart Bastian Koller Nobelstrasse 19 70569 Stuttgart Germany Tel: +49 711 68565891 E-mail: koller@hlrs.de	**University of York** Neil Audsley Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325571 E-mail: neil.audsley@cs.york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**WINGS ICT Solutions** Panagiotis Vlacheas 336 Syggrou Avenue 17673 Athens Greece Tel: +30 211 012 5223 E-mail: panvlah@wings-ict-solutions.eu

DOCUMENT CONTROL

Version	Status	Date
0.1	Definition of TOC	03/05/18
0.2	Integration of first inputs from EGM, WINGS and HLRS	29/06/2018
0.3	Integration of updated inputs from EGM, HLRS and Unparallel	27/07/2018
0.4	QA for EC delivery	03/08/2018
0.5	Internal QA updates from UoY and Intecs	10/08/2018
0.6	Overall updates from EGM, WINGS, HLRS and Unparallel	17/08/2018
1.0	Final Version	04/09/2018

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This document describes the final development of the PHANTOM tools and technologies, which are achieved by the Parallelization Toolset, Programming Interfaces, Model Based Testing and Monitoring Library components of the PHANTOM platform.

Section 2 describes the Parallelization Toolset. Technologies and algorithms are presented in this section for both the code analysis and the suitable techniques selection for the preparation of the components for deployment. In particular, this section describes the methodology used to produce parallelized versions of the components for CPUs and GPUs, as well as the adoption of specific parallelization techniques based on the deployment plan provided by the Multi-Objective Mapper. In the context of the Parallelization Toolset, it also summarizes the development work on the IP Core generator for FPGAs.

Section 3 introduces the PHANTOM Programming Interface to support the development of PHANTOM applications. This section identifies and describes a set of APIs following a component-based approach. These APIs use the C programming language and allow the use of generic parallelization functionalities, addressing both synchronization and data sharing mechanisms. They also provide support for any kind of deployment allowing the interchange between different deployment plans without the user's interference.

Section 4 reports the Model Based Testing (MBT). MBT in PHANTOM consists of two mains phases, i.e., early validation and test execution, for functional and non-functional testing. Early validation detects early design defects in parallel with application development. Test execution thoroughly examines the PHANTOM applications by executing test cases generated from MBT models. The main achievements of MBT in PHANTOM lie in the MBT extension to test applications in embedded and parallel environments and the specific strategies to test PHANTOM component-based applications. Both early validation and test execution produce promising results, while testing effectiveness and efficiency have been largely improved in PHANTOM.

Section 5 presents the Monitoring Library for application optimization based on non-functional properties and hardware quality attributes. The PHANTOM Monitoring Library abstracts users from the metric collection process. The major innovations are identified for the Monitoring Library are 1) administrator to register the monitoring configurations on heterogeneous systems, which frees the users of such task as well as the necessary expertise to do it; 2) the Monitoring Library offers the users a set of light-weight, hardware-agnostic APIs for injecting the instrumentation into the application code, which provides a tight integration of the Monitoring Framework with the user application; 3) highly customizable monitoring settings. The users can specify the metrics to be automatically collected, the flush time interval, the configuration of collected metrics, etc., independently for each application.

1. INTRODUCTION

1.1 RATIONALE AND MOTIVATION

In order to achieve the integrated cross-layer, multi-objective and cross-application approach in PHANTOM, a number of components are designed and developed in PHANTOM and the following four components are reported in this deliverable.

The Parallelization Toolset automatically parallelizes sequential application code. Besides the typical parallelization in a standard environment, the Parallelization Toolset is able to provide parallelized code for different environments including CPU, GPU and FPGA platforms.

The application components communicate using the PHANTOM Programming Interface included in the programming model. The transparency provided allows the developer to express their application as an interacting set of components that communicate over a common interface. Along with the adoption of the Programming Model, the interface creates an integrated environment for the application to run. By enforcing this model, the rest of the platform can operate on individual components by deploying them in different locations (or to GPU devices or FPGAs) simply by ensuring that the communications are correctly handled. All PHANTOM tools operate at the component level.

Instead of traditional testing methods in which test cases are manually developed, Model Based Testing (MBT) automatically generates test cases from MBT models. The test generation process follows user specified strategies, and this guarantees a systematic testing coverage to explore and test application behaviors when executing the test cases. In PHANTOM, besides the requirements to improve testing efficiency and effectiveness, one particular testing requirement is to enable programmers to test very early in the development life cycle. Following this, MBT in PHANTOM provides two early validation activities (i.e., model validation and performance estimation) in the design phase to test applications based on specification documents without executing binaries. Moreover, the PHANTOM MBT approach builds upon the abstractions brought by the PHANTOM Programming Model to focus on global functional and non-functional properties of the system, so early validation and test execution activities are designed and implemented for both functional and non-functional testing.

The Monitoring Library abstracts the collection of non-functional and user-defined metrics, automating the collection of optimisation data. The library allows the selection of the metrics to be collected at the system and application levels, as well the configuration of the sampling frequency. This mechanism is missing from existing commodity tools. The MF-Library was integrated with the MF-Server under task 2.2 "Unified runtime monitoring implementation". However, we consider that MF-Library API, targeting to enable the collection of metrics at application level in a user-defined manner, has to be described in D3.2 because is there where are described the PHANTOM APIs ("Programming interfaces (APIs) per application class" task 3.2). This allows having all the tools used for the instrumentation of the applications in one place, as well as helps to provide a clearer view on the purpose and future use of the MF-Library API.

The components in this deliverable, together with the components defined in WP2 and WP4, constitute the next generation heterogeneous, parallel and low-power computing systems in PHANTOM.

1.2 SCOPE

This document reports the development of all tasks executed in the context of WP3 – “Programmer- and productivity- oriented software tools”. Figure 1-1 shows a representation of the PHANTOM architecture. Highlighted in red are the components developed within the context of WP3 activities. These components are Parallelization Toolset, Programming Interface, Model Based Testing and PHANTOM Monitoring Library.

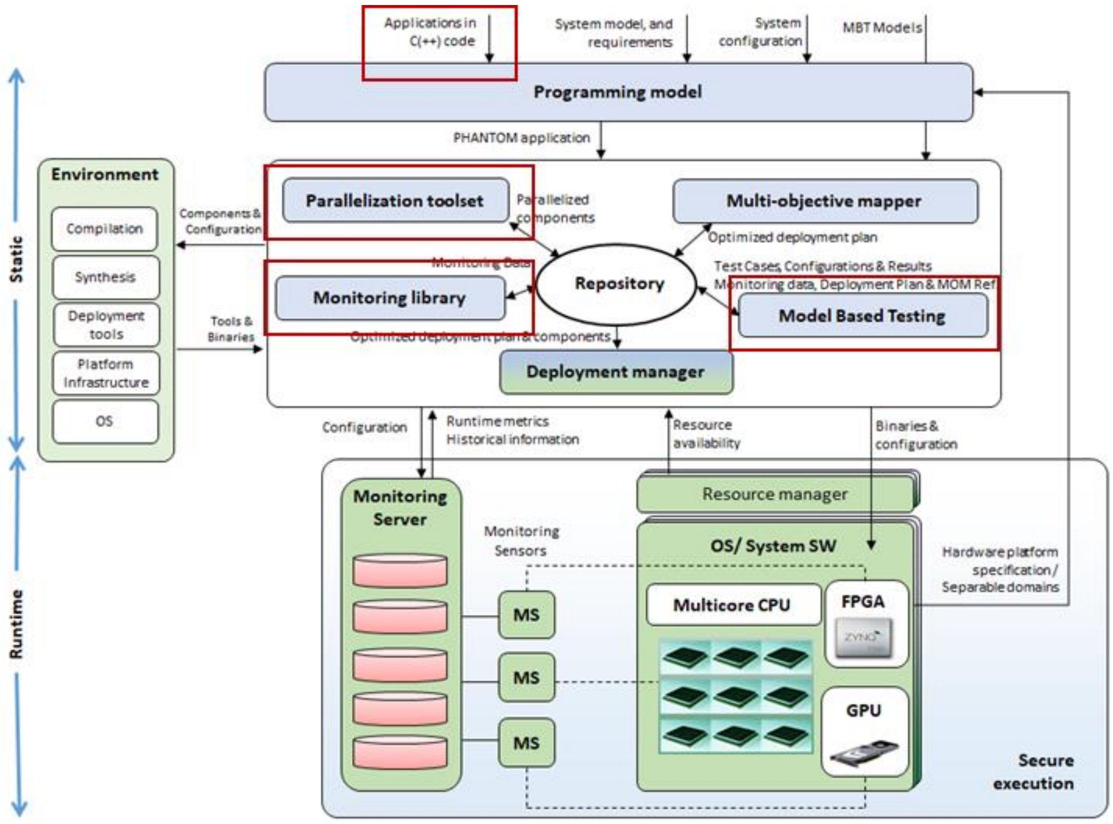


Figure 1-1: Components of the PHANTOM architecture addressed in WP3

1.3 AMBITIONS AND MAJOR INNOVATIONS

The Parallelization Toolset provides a first stage of translation for the application, meaning that it transforms the application so that a greater exploitation of the available hardware resources is achieved. It is also responsible for applying the necessary modifications that are going to enable the application’s deployment on different hardware devices like FPGAs or GPUs. In this way, new levels of code parallelization can be reached that will be able to overcome results from state-of-the-art techniques applied on homogeneous platforms. This is important since popular techniques on SMPs like *pthreads* or *OpenMP* can be gracefully combined with NUMA architectures, as well as execution on external acceleration devices.

The Programming Interface is an enabling technology designed to provide the communication and synchronization between the application components. The provided APIs can be used regardless of the background architecture that has been chosen (POSIX, MPI, …) without any user interaction providing to the final product an optimized execution environment.

MBT in PHANTOM advances the MBT technologies with the following innovations.

- MBT in PHANTOM enables early validation to check the functional and non-functional characteristics of a design in parallel with application development.
- MBT further improves testing efficiency and effectiveness with fine grained and adapted MBT components. Separate models are developed in the early validation and test execution phase to capture different testing focuses. MBT models are created based on the extension of state machine diagrams to consider parallel features of PHANTOM applications. The test cases are generated with systematic criteria to provide effective coverage and execution.
- MBT has been extended to test applications in parallel and embedded environments, while adaptive solution has been developed to efficiently test component-based applications. Based on the MBT achievements in PHANTOM, we have applied for an innovation patent.

The following three major innovations are identified for the Monitoring Library.

- It frees the end users from providing a hardware description or monitoring configuration on heterogeneous systems.
- It can monitor user-defined metrics for application-level monitoring without the need of additional monitoring tools, and offers a light-weight, hardware-agnostic API for injecting the instrumentation into the application code.
- It has highly customizable monitoring settings. The users can specify the metrics for collection, and can modify parameters like the sampling and the flush time intervals independently for each application and metric, which is not the case for the other, alternative approaches. The tool even allows to modify the monitoring configuration during the execution of the application.

2. PARALLELIZATION TOOLSET

2.1 USE CASE REQUIREMENTS

2.1.1 Initial Set of Requirements

Appendix 1 contains a per-requirement breakdown of the functionality of the Parallelization Toolset. In summary, all requirements we met, apart from U38 which was eliminated based on end user feedback.

2.1.2 Preliminary use case feedback from M18 results

The PT Code Analysis takes sequential code and transforms it in order to enable loop and task parallelization. The first version of the tool was able to identify all existing loops inside a portion of a components' source code and to parallelize some of the non-complex ones. PT Technique Selection was then used to select the adequate parallelization technique. This has provided important automation of parallelization tasks that would otherwise have to be done manually.

GMV was already able to apply it to achieve a small improvement of the execution time through parallelization of the source code; however, it was detected that the tool was only targeting a small portion of the code, so a larger improvement was expected from later versions. The Code Analysis functionality has had some major improvements compared to the preliminary results from M18 (as shown in section 2.3.5).

2.2 DESIGN SPECIFICATIONS

The tool flow of the Parallelization Toolset consists of two main components: Code Analysis and Technique Selection. The former runs an analysis to the components' code to identify the parallelizable regions in it and produce modified versions that can run on different parts of the available hardware (CPUs, GPUs, FPGAs). The latter takes up to choose the corresponding versions based on the MOM's decision and upload them on the Repository to be used by the Deployment Manager for the execution of the application. Their architecture and their interaction with the other PHANTOM components are described below.

2.2.1 Code Analysis

In general, Code Analysis is responsible for the first analysis of the components' code and the production of their modified versions that are described as:

- **OpenMP version:** Includes OpenMP annotations that indicate the usage of multiple threads for the component's execution in CPU platforms.

- **CUDA version:** A version including the CUDA API that communicates with the kernel of the GPU (the code that runs on the device) to enable GPU acceleration. This version is generated only when specified in the component network in terms of feasibility (optimality is decided by MOM).

- **FPGA version:** A version of the component specifically designed to enable execution on an FPGA device. This version is generated only when specified in the component network in terms of feasibility and its generation is part of the Technique Selection module by using a specific tool entitled as IP Core Generator (as described below).

The generation of these versions of the components will give the ability to PHANTOM to exploit the available hardware platform and accelerate the application's functionality. The flow of the tool is described in the steps below.

2.2.1.1 *Component Network Analysis*

The main activity of the tool is the automatic parallelization of the components and the libraries that are defined by the user. For the implementation of that interaction with the user, the tool gets input from the **component network XML file** which is downloaded from its location on the Repository, containing all the necessary information about the components (e.g. name, source file, external libraries used etc.) and stores that information using a set of classes designed to model the different components and the data transfers between them.

2.2.1.2 *Automatic parallelization*

After extracting the necessary information from the XML file, the tool can proceed with the parallelization of the code. For this purpose, another interaction with the Repository is required for downloading the project source files. Finally, the actual analysis of the code is invoked for all the components and the external libraries (only the ones that are defined in the Component Network) that are used, as described in the following steps.

Pre-processing with the Programming Model

Characteristics of the PHANTOM programming model assist the code analysis stage. In specific, the component isolation of the model means that there are data dependencies discovered during the analysis that don't affect the potential concurrency. For this reason, specific **annotations** (described in D1.3) of the programming model are used to extend the tool's "understanding" of the data flow and resolve many of those dependencies. For instance, the annotation *#pragma phantom static-vectors* is used to characterize a vector of static size, resolving in that way lots of the dependencies that occur due to the variable size of the vector.

In order to exploit these indications, a big part of the PT's Code Analysis functionality concerns the pre-processing of the code, aiming to resolve a lot of its existing dependences. The tool uses this interaction with the user to resolve some of the difficulties that the latest techniques in automatic parallelization have yet to resolve.

OpenMP Parallelization

Code Analysis identifies parallelizable regions in the code and produces a parallelized version, containing OpenMP annotations that allow the use of multiple threads during the execution of the component on a multicore CPU system. The generated parallelized versions are uploaded on the Repository.

CUDA API for deployment on GPU devices

As with the OpenMP version, Code Analysis identifies parallelizable regions in the code and produces modified versions, using CUDA functions to transfer data in and out of the device and invoke the kernel functions to execute the necessary calculations defined by the user. The new versions are uploaded on the Repository. For FPGAs, the Technique Selection is taking care of the IP Cores generation (through the IP Core Generator tool).

Component Network Update

According to the above results, Code Analysis updates the Component Network XML file by including parallelization directives for each component (ability to be parallelized, possible number of independent loop slices etc.).

Repository Updates and Application Manager Notification

After the analysis is completed, the updated Component Network XML file and the updated versions of the components are uploaded to the Repository. Finally, a notification is sent to the Application Manager, so that the MOM is informed that the analysis stage is over and that the updated version of the Component Network can be obtained by the Repository.

2.2.2 Technique Selection

CPU and GPU Parallelization

Now that components have been annotated with their potential parallelism, Technique Selection actually selects which to use. A Deployment Plan is produced by the MOM which determines which version of each component to deploy according to the hardware resources assigned to each component using a straight forward decision mechanism.

In particular, for the case that the component runs on a CPU the OpenMP version is used, enabling the possibility of running the component in parallel using multiple threads. On the other hand, if the MOM decides to use a GPU or an FPGA to accelerate the component's execution the CUDA version is chosen for the GPU case or the selection procedure is assigned to the IP Core Generator which is responsible for the FPGA version of the components.

IP Core Generator

The IP Core Generator allows the automatic creation of FPGA IP Cores for accelerating components. The procedure of the IP Core Generator can be summarised in the following steps:

Step 1. The IP Core Generator receives a notification about new deployment plans previously validated by the Offline MOM;

Step 2. Checks if there are components, in the deployment plans, to be deployed for FPGA architecture;

Step 3. Downloads the source files, for those components, from the Repository;

Step 4. Transforms the source code, using LLVM and Clang compiler and toolchain technologies, to expose the IP Core interfaces that are required for generating an IP core using Vivado HLS toolkit;

Step 5. The IP Core Generator will invoke Vivado HLS tools to analyse the transformed source code and to create the hardware descriptive files (HDL files) of the IP Core;

Step 6. When Vivado HLS finishes, the IP core files are compressed into a zip file and uploaded back into the Repository to be used by the other PHANTOM Platform components.

2.2.3 Background Tools

ROSE Compiler

The Parallelization Toolset uses the **ROSE Compiler** [1] during its code analysis on each component or external library that is defined by the user. ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale applications. In specific, the PT employs the **autoPar** tool [2] which is an implementation of automatic parallelization using OpenMP and can automatically insert OpenMP 3.0 directives into input serial C/C++ code.

LLVM and Clang compiler

The IP Core Generator uses the **LLVM suite** [3] and the **Clang compiler** [4] to parse and analyse the code identified to be implemented in hardware. These tools are used to define and execute the transformation rules required to transform the component code into code accepted by the Xilinx Vivado HLS.

Xilinx Vivado High-Level Synthesis

The **Xilinx Vivado High-Level Synthesis** (HLS) [5] is a tool used to generate hardware designs based on C, C++ or System C specifications. The IP Core Generator uses this tool to automatically generate the hardware designs equivalent to functions in a PHANTOM component, selected to be deployed on an FPGA hardware. The IP Core Generator must transform the code of a PHANTOM component, adapting the code and introducing low-level descriptions to enable Vivado HLS the generation of the IP Cores.

2.3 IMPLEMENTATION DETAILS

The above design is implemented with the flow that is described, using the individual tools to complete its functionality.

2.3.1 Code Analysis

Code Analysis is implemented in Java. The input of the Code Analysis is downloaded from the Repository and consists of the Component network XML document and the specified components' source code (currently C/C++). The analysis is an iterative process that analyses the components and any external libraries that are used, one by one.

A big part of the analysis concerns the pre-processing of the code according to user annotations as they are defined by the Programming Model in D1.3. In specific:

```
#pragma function no-side-effects
```

This pragma declares that a function can be considered as side-effect-free, meaning that it doesn't write in memory addresses outside its range. That enables the PT to ignore any dependencies originating from any calls to this function.

```
#pragma loop no-pointer-aliasing
```

This pragma can be used to declare that any pointer/array accesses in the following loop will not overlap with each other. This information is used to eliminate dependencies that occur because of indirect array accesses using values that are defined at run-time.

```
#pragma loop static-vectors
```

This pragma can be used to declare that any vector-class objects that exist in the loop retain their size during all iterations avoiding in this way dependences caused by accesses to unallocated memory addresses. Using this information, the PT considers any vectors appearing in the analyzed loop as static C arrays. This indication was found to be the most useful since the vector class is used a lot in the use cases.

After pre-processing the component, the PHANTOM Parallelization Toolset applies some of the latest compilation techniques [6] to model the source code. An extended direction matrix (EDM) dependence representation is used to cover non-common loop nests that surround only one of the two statements in order to handle non-perfectly nested loops. For array accesses within loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables. Introducing such models assists the localization of otherwise untraceable major parallelization capabilities.

The stages included in the analysis [1] are the following:

- Apply optional custom transformations based on input code semantics, such as converting tree traversals to loop iterations on memory pools.
- Normalize loops, including those using iterators.
- Find candidate array computation loops with canonical forms (for `omp for`) or loops and functions operating on individual elements (for `omp task`).
- For each candidate:
    - Skip the target if there are function calls without known semantics or side effects.
    - Call dependence analysis and liveness analysis.
    - Classify OpenMP variables (autoscoping), recognize references to the current element, and find order-independent write accesses.
    - Eliminate dependencies associated with autoscoped variables, those involving only the current elements, and output dependencies caused by order-independent write accesses.

- Insert the corresponding OpenMP constructs if no dependencies remain.
- Store the new parallelized OpenMP version locally

Finally, the pre-processing of the code is reversed and the new version is uploaded on the Repository.

2.3.2 CUDA API for deployment on GPU devices

As already mentioned, the GPU API that is described in D3.1 is used to produce the GPU versions of the components. In specific, the PT generates the CUDA API that is required for the deployment on the GPU device. The necessary inputs and outputs are exchanged between CPU and GPU and the initiation of the kernel is coordinated by the corresponding functions. An example of the API's exploitation is displayed below:

```
int *vec_add(int *a, int *b) {
    int c[10000];

    #pragma phantom kernel vec_add_CUDA kernelin a type=int
size=10000
     #pragma phantom kernel vec_add_CUDA kernelin b type=int
size=10000

    for(int i=0; i<10000; i++)
        c[i] = a[i] + b[i];

     #pragma phantom kernel vec_add_CUDA kernelout c type=int
size=10000

     return c;
}
```

Here, the use of the GPU API is visible, defining the arrays `a` and `b` as inputs and the array `c` as output for the kernel function `vec_add_CUDA()`. The results of the analysis and the generation of the GPU version of the component is displayed below:

```
//GPU usable
int *vec_add(int *a, int *b) {

     int c[10000];

    //Declare pointers for the GPU to use
     int *dev_a, *dev_b, *dev_c;

    //Allocate memory on the GPU
    cudaMalloc((void**)&dev_a, 10000*sizeof(int));
    cudaMalloc((void**)&dev_b, 10000*sizeof(int));
     cudaMalloc((void**)&dev_c, 10000*sizeof(int));

    //Copy the arrays a and b to the GPU
    cudaMemcpy( dev_a, a, 10000*sizeof(int), cudaMemcpyHostToDevice
);
     cudaMemcpy( dev_b, b, 10000*sizeof(int),
cudaMemcpyHostToDevice );

    //Launch the kernel with 100/128 thread blocks of 128 threads
    dim3 block(128);
    dim3 grid((10000+ 127)/128);
     vec_add_CUDA<<<grid, block>>>(dev_a, dev_b, dev_c, 10000);

    //Copy the array c back from the GPU to the CPU
     cudaMemcpy( c, dev_c, 10000*sizeof(int), cudaMemcpyDeviceTo-
Host);

    //Free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
     return c;
}
```

2.3.3 Technique Selection

As already mentioned, Technique Selection selects the suitable versions of the components for the deployment according to the Deployment Plan produced by the MOM. The process of this selection concerns the translation of the deployment plan from a lower-level hardware representation to a higher-level software representation of the components. In this way, the final version of the application is uploaded to the Repository and is ready to be refined and deployed by the Deployment Manager.

```
public void TechSel {
    boolean gpu_flag=false, fpga_flag=false;
    for(int i=0; i<Components.size(); i++) {
         if(Components.get(i).getLoops().size() == 0) {
             Components.get(i).setFinalVersion("OpenMP");
            continue;
         }
        else {
            for(int j=0; j<Components.get(i).getLoops().size(); j++)
{
                if(Components.get(i).getLoops().get(j).RunsOnGpu())
{
                    gpu_flag = true;
                }
                else if( Compo-
nents.get(i).getLoops().get(j).RunsOnFpga) {
                    fpga_flag = true;
                }
            }
        }
        if(gpu_flag && !fpga_flag) {
            Components.get(i).setFinalVersion("CUDA");
        }
        else if(!gpu_flag && fpga_flag) {
            Components.get(i).setFinalVersion("VHDL");
        }
    }
}
```

2.3.4 IP Core Generator

In some situations, the developer may want to test the performance of executing specific code on FPGA platforms. However, as he/she may lack the expertise on how to implement hardware designs, she may request the PHANTOM framework to automatically generate the corresponding designs.

To force the invocation of the IP Core Generator, the developer must specify, with the pragma below, which part of the component's code to that the developer would like to be transformed and deployed on the hardware of a FPGA platform by indicating the top function to be parallelised.

```
#pragma function generate-ipcore <TOP_FUNCTION_NAME>
```

This pragma indicates the interest of the developer in test performance of a hardware implementation of this code and, at the same time, provides the indication to the PHANTOM framework that this code respects the conditions for the code transformation. The developer must also ensure that the selected function does not use software constructs that Vivado HLS cannot support. Vivado HLS is one of the industry-leading high-level synthesis solutions, but it cannot translate everything.

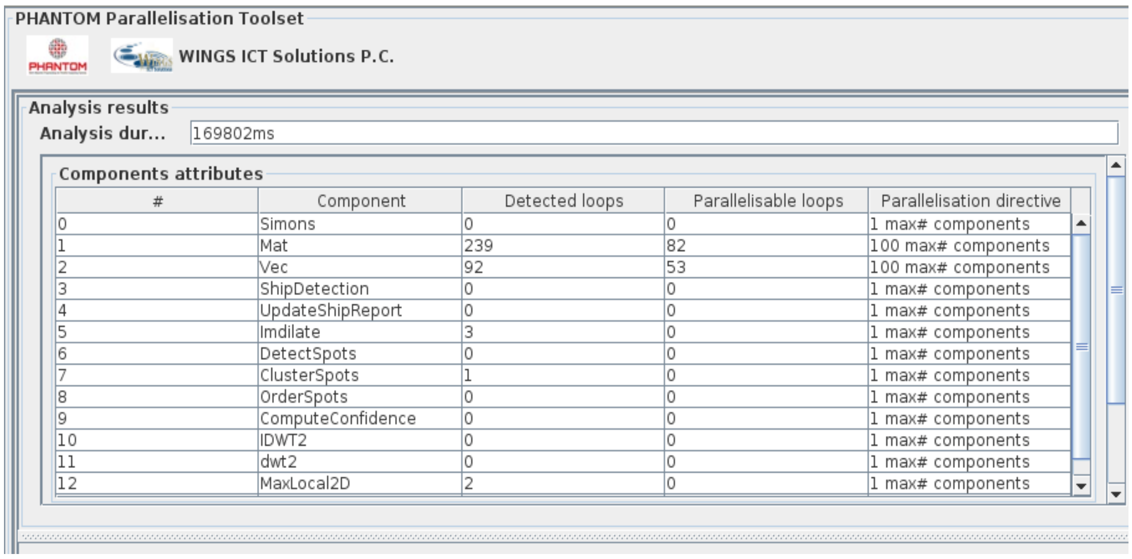
Though source code with variable dependencies can still generate a functional IP Core, static loop bounds and similar will generate more efficient hardware.

2.3.5 Preliminary testing results

The Code Analysis functionality has had some major improvements compared to the preliminary results from M18. In specific, the PT was able to analyze the component network reinforced with information about the external libraries that are used in the code to enlarge its focus on the parts that have the largest effect and extract revealing results.

The Surveillance use case was tested as shown in Figure 2-1 producing the following observations:

- The main components (which do not include the external libraries `Mat` and `Vec`) are not able to be parallelized since they are strongly characterized by their big size and complexity.
- Lots of the loops inside the `Mat` and `Vec` libraries that were successfully parallelized verify the tool's efficacy and expansion of those results after further integration with the use cases. In specific, we can see a very promising efficiency of **40% of parallelized loops** allowing many multithreading capabilities during the application's execution. Functions in the specified libraries are continuously invoked by the main components, hence their parallelization is massively optimizing the overall performance.



Analysis dur... 169802ms

#	Component	Detected loops	Parallelisable loops	Parallelisation directive
0	Simons	0	0	1 max# components
1	Mat	239	82	100 max# components
2	Vec	92	53	100 max# components
3	ShipDetection	0	0	1 max# components
4	UpdateShipReport	0	0	1 max# components
5	Imdilate	3	0	1 max# components
6	DetectSpots	0	0	1 max# components
7	ClusterSpots	1	0	1 max# components
8	OrderSpots	0	0	1 max# components
9	ComputeConfidence	0	0	1 max# components
10	IDWT2	0	0	1 max# components
11	dwt2	0	0	1 max# components
12	MaxLocal2D	2	0	1 max# components

Figure 2-1 Parallelization Toolset results

2.4 INTEGRATION ASPECTS

The Parallelization Toolset communicates with the other PHANTOM tools through the Repository and the Application Manager. The Repository stores all the files of every project, original or modified. The PT downloads the component network, component source code, and the deployment plan from the Repository, and uploads modified versions of the components and the component network.

The Application Manager implements the sending and receiving of notifications between PHANTOM tools. For example, a notification gets sent when a modified version of the Component Network is uploaded on the Repository, because the MOM is waiting for it to initiate its execution. Accordingly, the PT (Technique Selection) is polling, waiting for a notification from the MOM to inform the Application Manager that the Deployment Plan is uploaded and ready for the PT to use.

2.5 INNOVATIONS BEYOND THE STATE-OF-THE-ART

2.5.1 Summary of new technologies/extensions developed

2.5.1.1 XML Parsing and Modelling Classes

During the development of code analysis, the design of a set of classes and functions implemented in Java was found necessary, to parse and modify XML documents using appropriate libraries. These libraries are mainly required to process the Component Network that provides all necessary information about the components and the Deployment Plan that provides information about the deployment. All information extracted from these documents are modelled inside the tool providing a usable model that is currently able to guide the analysis of the components and the technique selection as well. These models are used to describe the Component Network and the Deployment Plan and can be used by any tools that require their usage (e.g. Deployment Manager).

2.5.1.2 Use of the PHANTOM Programming Model

The adoption of the **PHANTOM programming model** (concerning dependences as described in D1.3) faces some of the issues that the latest technologies in the science of automatic parallelization have yet to resolve. It enables the user to interact with the tool optimizing the results produced by their joint effort with the tool. This approach aims to ensure that the best techniques have been inspected with regard to the parallelization efficiency provided by the PHANTOM framework.

2.5.1.3 Integration and extension of external developing tools

The integration of ROSE with the PT has fitted well on the final structure of the tool resulting in the adoption of the latest technologies available, while also satisfying the constraints regarding the use case requirements. With support from the programming model, the PT is able to extend the functionality of the *autoPar* tool, eliminating lots of dependences that make real-world applications difficult to parallelize, while still using advanced mathematical models to conduct the code analysis.

In addition, the functionality of the PT tool flow is extended with the support of GPU-based or FPGA-based components resulting in a full-platform support, unlocking multiple parallelization capabilities. Deployment on multiple devices can be achieved while minimum user intervention is required, hence creating a tool that can guarantee full code coverage for multiple-type applications.

2.5.1.4 IP Core Generator

High-level Synthesis tools exist, but they still require a large amount of expert knowledge to integrate into an FPGA design. The componentisation of the PHANTOM programming model allows the IP Core Generator to automate the creation of IP cores in a much more automated way than is normally possible. Combined with the other PHANTOM tools which automatically combine IP cores into a fully supported design, this tool provides PHANTOM with the capability to fully automatically produce FPGA deployable artefacts without requiring any significative effort from the developer.

2.5.2 Full Prototypes functionality

The Parallelization Toolset, on its latest stage, is able to produce modified versions of the components supporting OpenMP for parallelization on a multicore CPU system as well as versions for deployment on GPUs and FPGAs. The tool is fully integrated with the PHANTOM Servers such as the Repository Server and the Application Manager Server. Being fully connected with those two components, the PT is able to cover all its integration needs with other PHANTOM components, allowing it to run when required, fetch the data it needs, and update the Repository with the generated results.

The PT adopts the functionality of the ROSE Compiler to perform deep analysis tests, identifying data dependencies in the components and producing their parallelized OpenMP versions. GPU versions are created as well, as products of the analysis with help from the GPU API that guides the transformation of the components.

Technique Selection uses the Deployment Plan provided by the MOM to choose between the different versions created to guarantee the deployment that the deep analysis conducted by the MOM is efficiently followed. The selection is guided by the Deployment Plan, producing a higher-level (software) representation of the final deployment, instead of the lower-level (hardware) one described by the MOM.

3. PROGRAMMING INTERFACE

For the required communication between the components, a set of APIs is provided by the PHANTOM framework (as described in deliverables D1.2, D1.3, D3.1) to enable the user to initiate and coordinate the data exchanges. In the case that the user would like to include execution of a GPU kernel for the acceleration of a component, a suitable GPU API is provided as well (as described in D3.1), to allow the user to coordinate the data transfers to and from the device and to initiate the deployment on it. The Programming Interface is defined as the implementation of the programming model communication protocols defined in D1.2, D1.3. The aforementioned APIs of the Programming Interface are described in detail in later paragraphs.

3.1 USE CASE REQUIREMENTS

3.1.1 Initial Set of Requirements

All preliminary use case requirements are met by the Programming Model. For a full breakdown, see Appendix 2.

3.1.2 Preliminary use case feedback from M18 results

The adoption of the PHANTOM programming model by the Telecom Use Case, with its multi-program paradigm, resulted in a cleaner partitioning and isolation of the modular and parallelizable components, thus in more structured and reusable code.

The Shared protocol model was adopted in the Telecom Use Case, by allowing a formal definition of the interaction mechanism between components and orientating the system design to a more secure communication approach. The model is completed by the formal description of its organization by means of the required platform description, component network, and the initial deployment plan.

3.2 DESIGN SPECIFICATIONS

3.2.1 Data Transfer Support

The PHANTOM Programming Interface provides an API to transfer data according to the needs of the application. Thus, all possible efforts have been made in order to extend the support of the data that can be moved among the different components.

Automatic Support: Primitive data types are all automatically supported by the Programming Interface. This includes the types:

- char
- short
- int
- long
- float
- double

and all qualified versions of the above (`signed`, `unsigned`, `long`, and `long long`, and those declared `const`).

Extended Support:

The Programming Interface also supports:

- Structs and unions of the above types
- Arrays of the above types (including supported structs and unions) that are declared with a compile-time static size. This includes multidimensional arrays, where all dimensions have a compile-time static size.
- Variable-length arrays of the above types, since their size can be manipulated by the `sizeof()` operator at compile-time.
- Class instances of the above
- References to the above

Precisely, an array with a compile-time static size are arrays where `sizeof()` can resolve to a value at compile-time without having to generate run-time code.

For the use of primitives, users are encouraged to use types included in the `stdint.h` library, because of their constant size among different architectures (as discussed in D1.2). In a different case, the Deployment Manager will take up to resolve type inconsistencies before the deployment (details in D4.4).

Pointer types are not automatically supported because they are not meaningful when transferred to another memory space. This restriction also applies to structs with pointer fields.

The same stands for C++ vectors and other structures whose size is variable and cannot be known at compile-time.

Handling Unsupported Situations: Cases where the data that needs to be transferred follows a more complex structure than the ones mentioned above, can also be easily handled by the user by including serialization of the data into a static structure (array) before the transfer like shown in the example below.

```
struct X *data_out; // Unsupported data type

int component_A() {
        initialize(data_out);

        #pragma phantom queue out object_out
        char object_out[100]; // Supported data type

        serialize_X(data_out,object_out);

        phantom_queue_put(object_out);

        return 0;
}
```

```
struct X *data_in; // Unsupported data type

int component_B() {
        #pragma phantom queue out object_in
        char object_in[100]; // Supported data type

        phantom_queue_get(object_in);

        deserialize_X(object_in,data_in);

        return 0;
}
```

In general, the Programming Interface provides to the user a wide support for all kinds of transfers. Attention should be paid to the limitations (described above) coming from the address space switching when moving objects among different parts of the hardware architecture.

3.2.2 Shared Protocol

The shared API consists of functions for manipulating data stored in the shared memory. As described in D1.2 PHANTOM provides the user with the ability to declare a variable as shared via the appropriate phantom directive (i.e. `#pragma phantom shared` etc.). Since PHANTOM does not provide automatic consistency, the developer must call synchronization functions in order to update the shared variable in its latter status. For this purpose, PHANTOM provides the following function:

`bool phantom_synchronize(void *item, int dir)` (1)

which causes the local view of the item to be updated according to the corresponding data on the shared memory if the dir variable has the value 0. On the opposite case, the shared memory is updated according to the local view of the item.

Furthermore, in case a component, along with its shared data, is parallelized in slices by the Multi-Objective Mapper, PHANTOM provides the following function:

`size_t phantom_slice_size(void *item)` (2)

able to return the size (in elements) of the slice allocated to this component. This function, when combined with the appropriate offset, can provide the data processing (e.g. iteration) only between the targeted offset and the returned slice size.

3.2.3 Queue Protocol

The Queue API offers the appropriate facilities that enable the user to manage the communication between components that are mainly mapped to distributed memories and are linked with specific communication objects in the form of queues. Specifically, these queues have the form of blocking FIFOs of arbitrary size (see D1.2). The functions of the Queue API provide the user with the ability to send or receive elements and to count the number or size of the elements which are in the queue. The Queue API functions addressed in the current stage are the following:

```
bool phantom_queue_get(void *item) (3)
```

```
bool phantom_queue_put(void *item) (4)
```

```
bool phantom_queue_peek(void *item) (5)
```

```
uint32_t phantom_queue_count(void *queue) (6)
```

3.2.4 Signal Protocol

Apart from sharing and interchanging elements, PHANTOM includes the Signal API which provides the user with signalling facilities, enabling the coordinated execution of components and sections inside each component, without sending or receiving queues or shared data. These Signal API functions also complement the functions of the Shared and Queue API since they also need signalling to synchronize their execution, between components. PHANTOM provides functions able to block and wait for specific signal, which are unblocked by appropriate notify functions, as described in D1.2:

```
bool phantom_wait(void *signal) (7)
```

Blocks the current thread/process until the signal in question is notified.

```
bool phantom_notify(void *signal) (8)
```

Unblock a random single thread/process waiting on the signal.

```
bool phantom_notifyall(void *signal) (9)
```

Unblock all threads/processes waiting on the signal. In addition, PHANTOM provides a barrier function able to wait until all threads or processes, before that call have finished their work:

```
bool phantom_barrier(int component_id) (10)
```

3.2.5 Mutex Protocol

Mutexes are used to enforce mutual exclusion between a set of components on a critical region where shared data is used. They are enforced at the component level and are unrelated to OS-level mutexes from other APIs, but their usage is similar to the one described in the POSIX standards. Applies to pointer and array types. The API functions that enable mutex usage are described below:

```
bool phantom_mutex_lock(void *mutex) (11)
```

Block until the current mutex can be owned by the requesting component.

```
bool phantom_mutex_unlock(void *mutex) (12)
```

Release the current mutex, if it is currently owned. Any components waiting on `phantom_mutex_lock` can then recontest for the mutex. If multiple components are blocked then a random one is awarded the lock.

```
bool phantom_mutex_trylock(void *mutex) (13)
```

Attempt to lock the mutex, but do not block if the attempt was unsuccessful. Returns true if the mutex was locked and false if not.

3.2.6 Repository File Access API

Feedback from user partners suggested that the PHANTOM Repository could be a useful resource for storing input and output data for the components of the application. In order to optimize the file access and minimize the communication cost created by it, data items are placed in the necessary locations throughout the deployment architecture. For example, large data items are placed 'near' the components that use them. Therefore, if components are able to read and write from the same memory that they are executed on, deployment over heterogeneous distributed architectures is optimized. This is achieved by extending the PHANTOM Programming Interface with file operations. The added functions are mirrors of the POSIX file operations in order to maximise compatibility. The stream objects returned by PHANTOM Programming Interface functions are compatible with the other I/O functions from the C standard library, `fprintf`, `fscanf`, `snprintf`, `sprintf`, and `sscanf`.

```
FILE * phantom_fopen ( const char * filename, const char *
mode ) (14)
```

Opens the file whose name is specified in the parameter filename and associates it with a stream that can be identified in future operations by the FILE pointer returned.

```
int phantom_fclose ( FILE * stream ) (15)
```

Closes the file associated with the stream and disassociates it. All internal buffers associated with the stream are disassociated from it and flushed: the content of any unwritten output buffer is written and the content of any unread input buffer is discarded. Even if the call fails, the stream passed as parameter will no longer be associated with the file nor its buffers.

```
int phantom_fflush ( FILE * stream ) (16)
```

If the given stream was open for writing any unwritten data in its output buffer is written to the file. The stream remains open.

```
size_t phantom_fwrite (const void * ptr,size_t size,size_t
count,FILE * stream) (17)
```

Writes an array of count elements, each one with a size of size bytes, from the block of memory pointed by ptr to the current position in the stream. The total number of elements successfully written is returned. Sets ferror if an error occurred.

```
size_t phantom_fread ( void * ptr, size_t size, size_t
count, FILE * stream ) (18)
```

Reads an array of count elements, each one with a size of size bytes, from the stream and stores them in the block of memory specified by ptr.

```
int phantom_fgetpos ( FILE * stream, fpos_t * pos ) (19)
```

Retrieves the current position in the stream.

```
int phantom_fseek ( FILE * stream, long int offset, int
origin ) (20)
```

Sets the position indicator associated with the stream to a new position. If the stream is binary, the new position is offset + origin. If the stream is text, then offset shall be 0. Returns zero on success or a platform-specific error number.

3.3 Implementation Details

Depending on the type of physical memory that is required for the communication, the protocol uses different libraries as described below.

In the case that all the components that interact with the communication object are executed on a single shared memory system, the specific components are going to be executed as different threads sharing the same memory space. To achieve that, PHANTOM uses pthreads along with other POSIX functionalities (like mutexes and condition variables) to keep the communication cost low as well as the memory footprint.

In the case that the components that interact with the object run on different nodes, meaning a distributed memory system, a Message Passing technique is required. For this reason the OpenMPI library was chosen due to its efficiency and simplicity on different kinds of real world applications. In specific, the execution of different components will be implemented as different MPI processes and the communication between them will exploit a lot of the already existing OpenMPI interface functions that are available.

In the following paragraphs, a description of the protocols implementation' is attached along with some of the key functions of the protocols, in order to display a good view of the Interface's functionality. More details about the implementation will be provided in D4.4, due to the major dependence of the Programming Interface on the Deployment Manager.

3.3.1 Shared Protocol

The Shared Protocol is implemented by keeping an area in the available memory as a shared segment between the different components.

```
if(dir == 0) {
    pthread_mutex_lock(&obj->lock);
    memcpy(local_data,shared_data,dims[0]*data_size);
    pthread_mutex_unlock(&obj->lock);
}
else if(dir == 1) {
    pthread_mutex_lock(&obj->lock);
    memcpy(shared_data,local_data,dims[0]*data_size);
    pthread_mutex_unlock(&obj->lock);
}
```

When executing on different nodes, the functions *MPI_Win_allocate_shared()* and *MPI_Win_shared_query()* are used to create a virtually shared area in the processes' shared address space, presenting to the user the image of a shared memory system. Below, the creation of such an environment is displayed, along with the implementation of the the *phantom_synchronize()* API. The generation of the displayed *main* function is part of the Deployment Manager's functionality and will be fully discussed in D4.4. Here, the **allocation** of the required shared memory segment is presented, as well as the necessary function to **update** the local memory with the shared version of the data (or vice versa).

```
int main(int argc, char** argv)
{
    printf("Initializing...\n");
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    MPI_Comm_group(MPI_COMM_WORLD,&world_group);
    MPI_Group_incl(world_group, 2, shared_ranks0, &shared_group0);
    MPI_Comm_create(MPI_COMM_WORLD, shared_group0, &shared_comm0);
    shared_size0 = -1;
    shared_rank0 = -1;
    if(shared_comm0 != MPI_COMM_NULL) {
    MPI_Comm_size(shared_comm0, &shared_size0);
    MPI_Comm_rank(shared_comm0, &shared_rank0);
     // Creation of shared memory window

    MPI_Win_allocate_shared(2*5*sizeof(float), sizeof(float),
MPI_INFO_NULL, shared_comm0, &local_var0, &shared_win0);
          }

    printf("Finalizing...\n");
    MPI_Comm_free(&shared_comm0);
    MPI_Group_free(&shared_group0);
    MPI_Group_free(&world_group);

    MPI_Finalize();
}
```

```
void synchronize(MPI_Win shared_win, MPI_Aint* winsize, int*
windisp, void* shared_var)
{
    MPI_Win_shared_query(shared_win, 0, winsize, windisp,
shared_var);
}
```

3.3.2 Queue Protocol

The Queue Protocol uses a shared memory area as well to keep the data of the queue. In addition, a common pointer variable is used to access the queue and manipulate its data.

```
void phantom_queue_get(phantom_queue *queue, void* it, int data_size)
{
	while(queue->count == 0);
	pthread_mutex_lock(&queue->lock);
	phantomQnode *tmp = queue->front;
	queue->front = queue->front->next;
	if(queue->front == NULL)
		queue->rear = NULL;
	queue->count--;
	memcpy(it,tmp->item,data_size);
	pthread_mutex_unlock(&queue->lock);
	free(tmp);
}
```

```
bool phantom_queue_put(phantom_queue *queue, void* it) {
	phantomQnode *new_node = generate_phantom_Qnode(it);
	pthread_mutex_lock(&queue->lock);
	if(queue->rear == NULL) {
		queue->front = queue->rear = new_node;
	}
	else {
		queue->rear->next = new_node;
		queue->rear = new_node;
	}
	queue->count++;
	pthread_mutex_unlock(&queue->lock);
	return true;
}
```

In order to translate the same functionality to a distributed memory system, the implementation uses the same logic as the shared protocol along with the MPI_Send(), MPI_Recv() functions to send and receive data.

3.3.3 Signal Protocol

The Signal Protocol uses POSIX conditional variables to transfer signals between components.

```
void phantom_notify(phantom_signal *curSignal, bool *rd, int phan-
tom_src) {
    pthread_mutex_lock(&curSignal->lock);

    curSignal->ready = *rd;
    if(curSignal->ready == true) {
       if(pthread_cond_signal(&curSignal->cond) != 0) {
              fprintf(stderr,"Failed to send signal\n");
              exit(0);
       }
    }
    else {
        perror("Signal called with value 0");
        exit(1);
    }
    pthread_mutex_unlock(&curSignal->lock);
}
```

```
void  phantom_wait(phantom_signal  *curSignal,  bool  *rd,  int  phan-
tom_cmpid) {
     pthread_mutex_lock(&curSignal->lock);
     while(curSignal->ready == 0) {
     if(pthread_cond_wait(&curSignal->cond,&curSignal->lock) != 0) {
         fprintf(stderr, "failed to wait the condition variable\n");
         exit(0);
         }
          else {
            curSignal->ready = 1;
            *rd = true;
          }
        }
        pthread_mutex_unlock(&curSignal->lock);
}
```

When executing on different nodes the MPI_Send(), MPI_Recv(), MPI_Bcast() functions are used for signalling and MPI_Barrier() for the phantom_barrier() PHANTOM function.

```
void phantom_notify(int *rd, int dst) {
      MPI_Send(rd, 1, MPI_INT, dst, 0, MPI_COMM_WORLD);
}
```

```
void phantom_wait(int *rd, int src) {
      MPI_Status *status;
      MPI_Recv(rd, 1, MPI_INT, src, MPI_ANY_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
```

3.3.4 Mutex Protocol

The Mutex Protocol uses the corresponding functionality provided by the POSIX standards. When executing on different nodes an approach similar to the Signal Protocol is considered using the `MPI_Send()`, `MPI_Recv()` functions.

3.3.5 Repository File Access API

For accessing files on the Repository, the API is used as an interface for the Deployment Manager which is responsible to download the corresponding file and place it on the same node that the component is executed. From there, the component accesses the file by using the File Access APIs that are mapped to the corresponding standard C library functions. In this way, the Deployment Manager creates a hidden layer that hides the intermediate transfers and virtualizes the Repository as a local file storage space.

3.4 INTEGRATION ASPECTS

3.4.1 Integration with the Deployment Manager

The PHANTOM Programming Interface is closely bound with the functionality of the Deployment Manager (which will be described in D4.4) as the latter is responsible for updating the code according to the deployment plan. In specific, the API calls are modified accordingly to include information about the actual API function implementations. The exact modifications will be described along with the full functionality of the Deployment Manager. Additionally, the necessary files are generated by the Deployment Manager in order to integrate the functionality of the application components and coordinate the communication protocols.

Specifically, for the File Access API, the Deployment Manager communicates with the Repository to download and place the necessary inputs on the corresponding location/node or to upload the application outputs on the Repository.

3.5 INNOVATIONS BEYOND THE STATE-OF-THE-ART

3.5.1 Summary of new technologies/extensions developed

3.5.1.1 Programming Model Support

The driving principle behind the PHANTOM Programming Model is that, if followed, it allows components to be moved automatically between deployment targets by the Deployment Manager. It can be implemented with a wide variety of backend technologies, from POSIX to CUDA to MPI, to support a wide range of implementation targets.

The Programming Interface is primarily focused on supporting coordination and data sharing between components, while still working in an environment such as the one that is described above. The provided APIs use the C programming language, enabling the incorporation of parallelization APIs (pthreads, OpenMPI, OpenMP and CUDA also described in D4.1), while also providing an abstraction of the system architecture, hiding the complexity between hardware and applications.

3.5.2 Full Prototypes functionality

The Programming Interface, in its full functionality, is responsible to satisfy the user’s needs for data exchanging and at the same time be abstract enough for the user to be able to manipulate data across the component network transparently of the deployment plan generated by the MOM. It also provides to the user an API qualified enough to be able to coordinate these aforementioned data transactions, guaranteeing concurrency. This is achieved by a hidden layer created by the Deployment Manager (described in D4.4) which enables the connection between the application and the actual implementation of the APIs.

The protocols that are described in the previous paragraphs are fully developed with regard to the user needs. The implementation of the Programming Interface is developed with regard to flexibility and simplicity for the user. Additionally, non-functional system requirements are faced with the selection of low-communication-cost libraries (e.g. pthreads) when applicable. The design of the functions also considers the memory footprint that is used in order to optimize the memory transfers needed for the execution.

4. MODEL BASED TESTING

Model-Based Testing (MBT) designates any kind of testing based on or involving models. Models represent the system under test (SUT), its environment, or the test itself, which directly supports test analysis, planning, control, implementation, execution and reporting activities. Besides the general list of MBT benefits [7] such as complexity management, debug support, and so on, MBT in PHANTOM further advances the design and implementation by deepening the following two testing advantages.

- **Enabling early testing.** Since MBT are based on models, analyzing and simulating models can provide insights of the intended implementation and detects potential defects so as to eliminate defects in design phrase and prevent defects from escalating to code level. The process does not involve the execution of real applications, so that early testing is enabled in parallel with application development.

- **Improving testing effectiveness and efficiency.** MBT formalizes testing-related activities as models and automates as many activities as possible to improve testing efficiency and effectiveness. Instead of writing a test case specification with hundreds of pages, test cases are automatically generated from MBT models following different criteria. The automatic test generation does not only improve the testing efficiency but the testing effectiveness as well by providing a systematic testing coverage for SUT.

Furthermore, compared to typical MBT activities in standard environments, the MBT has been extended to the following three fields in PHANTOM with specific testing strategies corresponding to each field.

- **MBT in embedded environments**. MBT for embedded is challenging due to the heterogeneity and connectivity of embedded systems [8]. In PHANTOM, system adapters have been developed to support the test execution in individual embedded environments, as well as the PHANTOM integrated embedded environment. System adapters provide communication channels between SUTs and MBT components. As far as relevant system adapters are available, MBT is able to execute other test cases reusing the system adapters over specific environments.

- **MBT in parallel environments**. PHANTOM's parallel environment enables the parallel execution of components and the communications between components of an application. In order to take this into account, the models developed in MBT must be adapted to handle concurrency. In PHANTOM, we have developed MBT models based on communicative state machine to capture both dynamic behaviors of components and the communications within an application, and thus the test case generated from the model are adapted to the parallel environment as well.

- **MBT for component-based applications**. PHANTOM applications are composed by internal components with specific patterns to enable control and data flow. Taking advantage of the component-based feature, we have developed two MBT activities so analyze functional flow and non-functional dependency among components as to enable early functional and non-functional testing without executing the application.

The objective of MBT in PHANTOM is to carry out black box testing for both use case applications and individual components of applications on the PHANTOM platform with a focus on global functional and non-functional properties of distributed and parallel computing environments. The SUTs are thus PHANTOM applications within PHANTOM computing environments along with the applications' components. **Functional properties** test that the SUT is able to produce the expected outputs when given the corresponding inputs under specific configurations. **Non-functional** properties refer to the SUT performance indicators such as execution time and energy consumption monitored by PHANTOM platform.

Concretely, the MBT in PHANTOM are conducted in two phases - early validation and test execution - along with four activities - model validation, performance estimation, functional testing and non-functional testing. Early validation is realized by model validation and performance estimation to check the functionalities of the intended implementations and estimate the performance in parallel with the application development, whilst test execution provides and executes concrete tests to conduct thorough functional and non-functional testing. The details are presented in the following sub sections.

4.1 USE CASE REQUIREMENTS

4.1.1 Initial Set of Requirements

MBT is applied in all three PHANTOM use case applications for quality assurance of functionality and performance. The initial set of requirements is presented in Appendix 3, and all requirements are met.

In addition to the general requirements on PHANTOM platform, specific use case requirements related testing are also identified by each use case in terms of specification documents. The three use case applications, i.e., Surveillance use case by GMV, Telecom use case by Intecs and HPC use case by HLRS, as well as their internal components in PHANTOM are the main SUTs of MBT. The specifications describe the expected functions and performance, which are presented in Appendix 4.

4.1.2 Preliminary use case feedback from M18 results

The first cycle of MBT has been conducted in the second year of the project and reported in D3.1, the focus of which are the model validation and functional testing for all three use case applications in individual environment (CPU and FPGA). Following that, the MBT work had been evaluated in the three parts: a) requirement coverage by use case partners, b) RRL by R&D partners, and c) KPIs of MBT by use case partners, as identified in D5.1, with the following preliminary feedback.

- **Requirement coverage**: requirements U29, U30 and U31 are respectively "somewhat covered", "somewhat covered" and "not yet covered" from requirement coverage evaluation. The main reasons for U29 and U30 are MBT are conducted in individual environments instead of a fully integrated heterogeneous HW targets, and the testing on integrated PHANTOM HWs has been done in the third year; the requirement coverage for U31 is "not yet covered" because the testing was mainly achieved by testers with extra development efforts to support test execution; in the third year we have provided two different APIs with use case partners to develop test cases, nevertheless U31 is a requirement with low priority.

- **RRL**: the overall reuse readiness level of MBT was 4 at the end of second year of the project; the improvement has been made in dimensions of documentation, intellectual property and support during the third year.

- **KPIs of MBT**: the MBT related KPIs are KPI 3.1 to 3.4 defined in D5.1, and the overall feedback about testing efficiency and effectiveness are satisfactory along with a high-level testing coverage of testing requirements.

At the end of the third year, a final evaluation of MBT will be realized and the validation results will be reported in D5.4 on M36.

Besides the preliminary feedback from use cases proving promising MBT results, some specific requirements and expectations have also been collected based on the MBT results in the second year. The specific requirements are illustrated as follows along with the specific tackling actions in the third year:

- **MBT for updated use cases**: use case applications have been improved based on previous testing results and have been updated with additional scenarios, components and functionalities. MBT in the third year has focused on the updated three use case applications.

- **MBT for integrated PHANTOM environments:** Along with the integration of use cases and different PHANTOM components with HWs, use case applications on PHANTOM heterogenous hardware with integrated interfaces needed to be tested for global quality assurance. MBT in the third years has developed extra components to support test execution in PHANTOM integrated environment.

- **Early non-functional testing**: model validation enabling early functional testing to detect defects proves to be effective. In the meantime, early testing for performance is expected to test non-functional properties of an application at the design phrase to validate the non-functional properties of intended implementation. We have implemented the performance estimation for early non-functional testing by considering the component-based feature and previous testing results of individual component.

4.2 DESIGN SPECIFICATIONS

Concretely, we have designed the MBT in PHANTOM as four testing activities (1 to 4 in Figure 4-1) in two main phases (I and II in Figure 4-1) to test both functional and non-functional properties of the SUT. Figure 4-1 lists the MBT activities in PHANTOM and the alignment with PHANTOM stages identified in D1.3, while the details are introduced below.

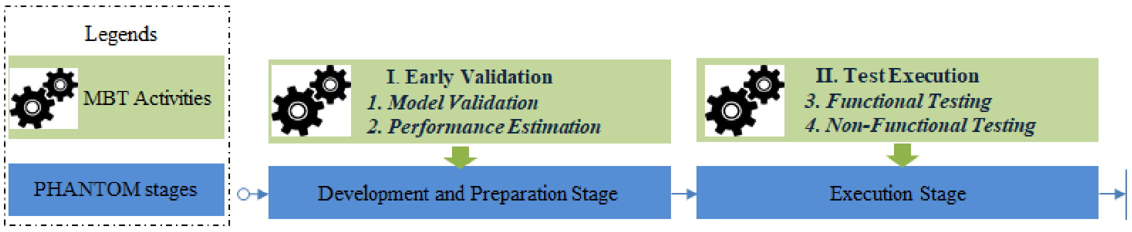


Figure 4-1 MBT activities in PHANTOM

4.2.1 Early Validation

Early validation is the first MBT phrase in PHANTOM, which tests and validates the design of an application's functional and non-functional properties. Early validation only relies on design specifications but does not require the execution of applications, and it is an early testing phase in parallel with the application development to detect design defects for follow-up correction and prevent them from escalating to implementation.

Specifically, the following two MBT activities - model validation and performance estimation - are conducted during early validation in parallel with PHANTOM development and preparation stage of applications. Model validation tests the functional aspect of applications, while performance estimation covers the application's non-functional performance.

4.2.1.1 Model validation

The model validation simulates the MBT models to check if the intended implementation contains any functional defects such as deadlock or over-designing (parts of the model never activated) and provides a summary for all detected functional defects. The corresponding workflow is illustrated in Figure 4-2.

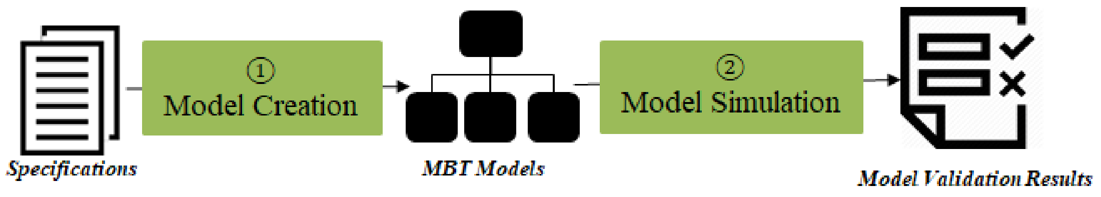


Figure 4-2 Model validation workflow

In all figures in section 4.2, the green rectangles represent MBT steps in one activity. The start and end point of an arrow represents inputs and outputs of this step, and the start and end point of the whole workflow is the global inputs and outputs of this MBT activity.

Step 1. Model Creation. In the first step, we create MBT models from use case specifications. The specifications define the testing requirements or the aspects to test of SUT (e.g., functions, behaviours and performances). The created MBT models represent high-level abstractions of SUT and are described by formal languages or notations such as UML, PetriNet and BMPN. In PHANTOM, communicative state machine is used as the meta model for the creation of MBT models to consider the communication of application components.

- Inputs: Specifications
- Outputs: MBT models

Step 2. Model Simulation. In this step, simulation scenarios with necessary data are automatically generated from MBT models and used to simulate MBT models. During model simulation, MBT models are validated regarding whether, under what conditions, and in which ways a part of model could fail to produce the correct outputs, and the corresponding deadlocks and over-designing are recorded.

- Inputs: MBT models
- Outputs: Model validation results

4.2.1.2 Performance estimation

Performance estimation estimates the non-functional properties (e.g., execution time, energy consumption) of newly designed applications by considering PHANTOM component network and previous non-functional testing results. PHANTOM applications are composed of components, and a new application can be easily created to recompose existing components in a different way for different tasks, such as the HPC scenario to simulate different topological structures with same simulation components. If all the inner components of the new application are previously tested with performance information, the performance of the application is than deduced by use of the previous testing results and their composition patterns. The corresponding workflow is illustrated in Figure 4-3.

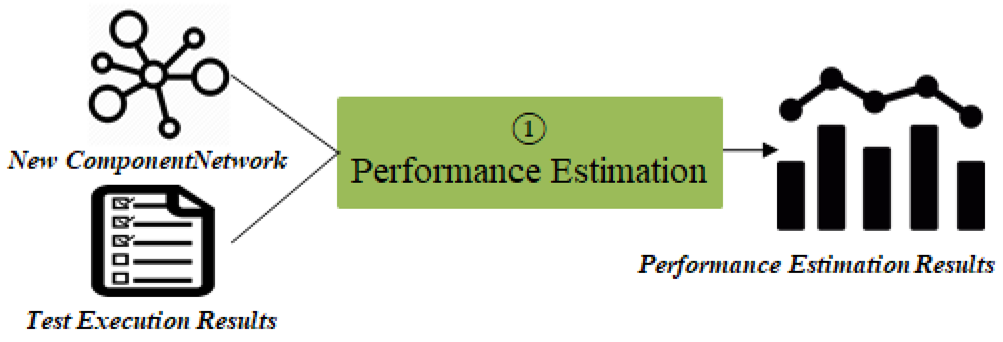


Figure 4-3 Performance estimation workflow

Step 1. Performance Estimation. In PHANTOM, each application is described by an xml file of component network indicating the internal components of an application and the patterns how the components are composed together. When given a component network description of an application, this step identifies the composition patterns among inner components, estimates the performance of the application based on an estimation model related to each non-functional property, and provides the estimation results.

- Inputs: Component network description and previous test execution results.
- Outputs: Performance estimation results.

This is particularly useful to estimate the performance of applications that reuse existing components and compose them in different ways to complete other computing tasks, such as HPC to simulate different topological structures with same components.

4.2.2 Test Execution

Test execution is the second MBT phrase in PHANTOM to execute concrete test cases against the SUT and collects thorough testing results for both functional and non-functional properties. Testing results are sent back to developers for further analysis and improvement. During the whole test execution phrase, traceability is kept between SUT specifications, test cases and testing results, so that developers can easily trace the defects sources from testing results to design specifications, and correct the defects based on the testing results.

The corresponding workflow of functional and non-functional testing activities is illustrated in Figure 4-4. Both functional testing and non-functional testing shares the same general workflow. However, since the functional testing and non-functional testing have different testing aspects, the MBT models and the executed test cases for the two activities are different from one another. The MBT models and test cases of functional testing focuses the inputs/output data flow for functionality, while the ones for non-functional testing collects performance information during the test execution and checks if the SUT meets the performance criteria.

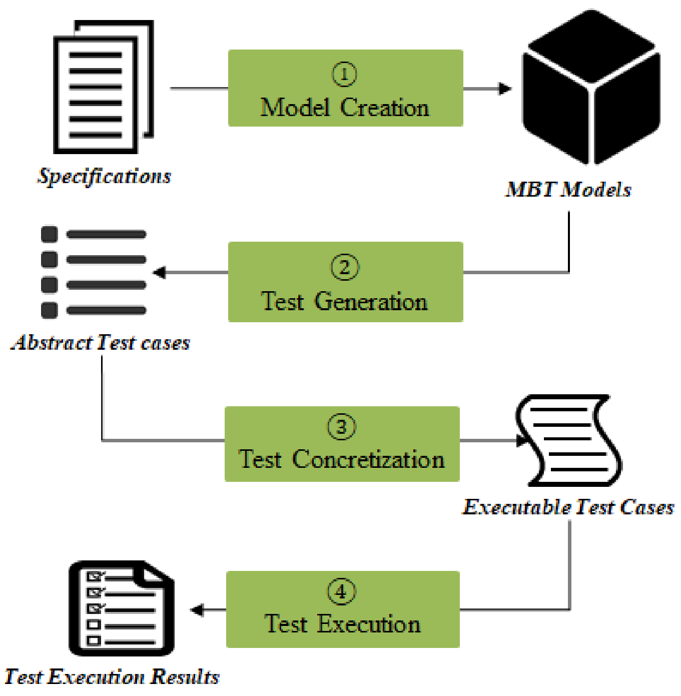


Figure 4-4 Functional and non-functional testing workflow

Step 1. Model Creation. In the first step, we create MBT models from use case specifications for test generation purpose.

- Inputs: Specifications
- Outputs: MBT models

The MBT models for test generation are different from the MBT models for validation in the following two points. Firstly, the MBT models for test generation do not only include the SUT's functional aspect of dynamic behaviours, but also contain the non-functional testing state and transitions to collect performance information. Secondly, the MBT models in the test execution phase focus on global inputs and outputs, which are different from MBT models in early validation phases with internal control and data details. This is because in order to provide a comprehensive validation result, the MBT models in early validation phase are expected to contain both internal data/control flows of components within an application as well as the global input-output relations between an application and its environment. Since the early validation is in parallel with development and does not impose timing constraints, a detailed model is important to largely exploit the model behaviours and detect early stage defects. On the other hand, the MBT models in test execution phase for test generation simply contains necessary input-output relations, as only inputs and outputs information are used to decide the pass/fail of a test, while internal communication is not captured during application execution, which also helps to improve testing efficiency.

Step 2. Test Generation. The second step automatically generates abstract test cases from MBT models when applying the test selection criteria. Test selection criteria guide the generation process by indicating the interesting focus to test, such as certain functions of the SUT or the structure of the MBT model (e.g. state coverage, transition coverage and data flow coverage). This process is typically automated by tools with test selection criteria options corresponding testing requirements.

- Input: MBT models
- Output: abstract test cases

Step 3. Concretization of Test Cases. The third step concretizes the abstract test cases from step 2 to executable test cases with mappings between the abstraction in MBT models and system implementation details. Executable test cases contain low-level implementation details and can be directly executed against the SUT.

- Input: abstract test cases
- Output: concrete test cases

Step 4. Execution of Test Cases. The executable test cases are automatically executed against the SUT. During the execution, the SUT is provided with inputs from each test case, and the outputs (for functional testing) and performance information (for non-functional testing) are collected to generate test verdicts.

- Input: executable test cases, system adapters
- Output: test verdicts, non-functional information

The four steps are performed iteratively and incrementally throughout development. The MBT process in PHANTOM implements this iterative and incremental approach, helping to guarantee full alignment with the test objectives and to keep MBT modelling activities efficient. The MBT process thus starts in parallel with the application development and assists the developers through the entire development process.

4.2.2.1 Functional testing

Functional testing automatically generates test cases from MBT models, stimulates the SUT in the PHANTOM environment with different test cases with input data, and compares the observed output with the expected output to decide if the tests pass or fail. In case that a test fails, feedback is reported to developers for correction and improvement.

4.2.2.2 Non-functional testing

Following the same workflow as functional testing, non-functional testing tests the performance of the SUT by executing the non-functional test cases generated from MBT models. At the end of test execution, non-functional properties are collected to provide information to other PHANTOM components (i.e. the MOM) as an initial mapping reference between application components and hardware, and MBT performance estimation to evaluate the newly design application’s performance. In the meantime, the obtained values of non-functional properties are checked with the non-functional requirements to decide if the SUT performance is satisfactory.

4.3 IMPLEMENTATION DETAILS

We have implemented and conducted all the four MBT activities (i.e., model validation, performance estimation, functional testing and non-functional testing) in two phases (i.e., early validation and test execution) for all three use case applications in PHANTOM. All testing results have been reported to developers. All MBT activities have been performed iteratively and incrementally starting with a preliminary specification for early validation and an early prototype for test execution. Any preliminary testing results were sent back to developers for application updates. With the further advance of the design and development of applications, MBT activities are correspondently updated to take into account new features. The MBT process in PHANTOM implements this iterative and incremental approach, helping to guarantee full alignment with the test objectives and to keep MBT modelling activities efficient. The MBT process starts in parallel with the application development and assists the developers through the entire development process.

In order to realize the MBT workflow for each activity, a number of MBT components are developed. The implementation details along with testing results are introduced in the following sub-sections.

4.3.1 Model validation

In model validation, we have developed the MBT models for each use case application and simulated the MBT model by DIVERSITY tool [10] to detect deadlock or over designing. Figure 4-5 shows the implementation details to achieve model validation.

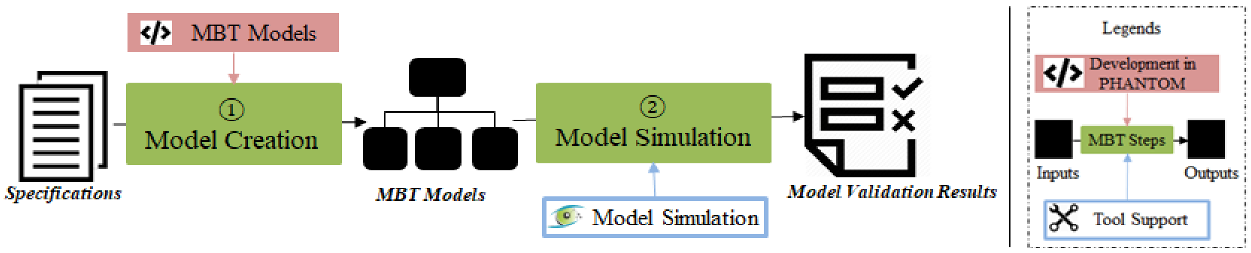


Figure 4-5 Model validation implementation

In all figures in section 4.3, the green rectangles represent the MBT steps in each MBT activity. The pink rectangles associated with green rectangles represent the MBT components we developed in PHANTOM to achieve this MBT step, and the blue rectangles represents the tool we use to support this MBT step.

MBT models. For each use case application, we developed a communicative state machine MBT model based on the case's specifications for simulation purpose. Individual models are first created for each component and then another model for the whole application are created to reuse and combine all individual component models. This mirrors how components are composed within the application. The validation of this MBT model allows a thorough exploitation of an application's behaviours during model validation.

In PHANTOM implementation, MBT models are created based on the metamodel "communicative state machine", communicative state machine is an extension of UML state machine, which models each individual application component as a state machine along with ports among state machines to enable the information exchange. This is particularly important in PHNATOM to take into account the communications among internal components and parallel architecture in PHANTOM.

To create the MBT models based on communicative state machines, we adopt the xLIA language due to the variety of primitives and the support of encoding all classical semantics. The MBT models are created in xLIA language within a textual environment. The following models for model validation are developed in PHANTOM:

- Telecom models for model validation
- HPC models for model validation
- Surveillance models for model validation

Although the models are created within a textual environment based on xLIA, we illustrate in the graphical visualization of the Telecom model as an example. In the figure, we present a global view of the model in the first half while a zoom view with details in the second half.

Model simulation tool. The model simulation is achieved by DIVERSITY. DIVERSITY is an open-source Eclipse based tool for formal analysis. We use DIVERSITY for model simulation purpose due to its support for symbolic execution. Symbolic execution uses symbolic parameters to represent simulation inputs rather than concrete numerical values so that multiple scenarios can be evaluated at the same time to simulate the models and explore the model behaviours more efficiently. Figure 4-7 illustrates the model simulation result for the Telecom use case, in which the left part shows the simulation process has covered all dynamic behaviours of the model, and the right part indicates that no deadlock or overdesign is detected.

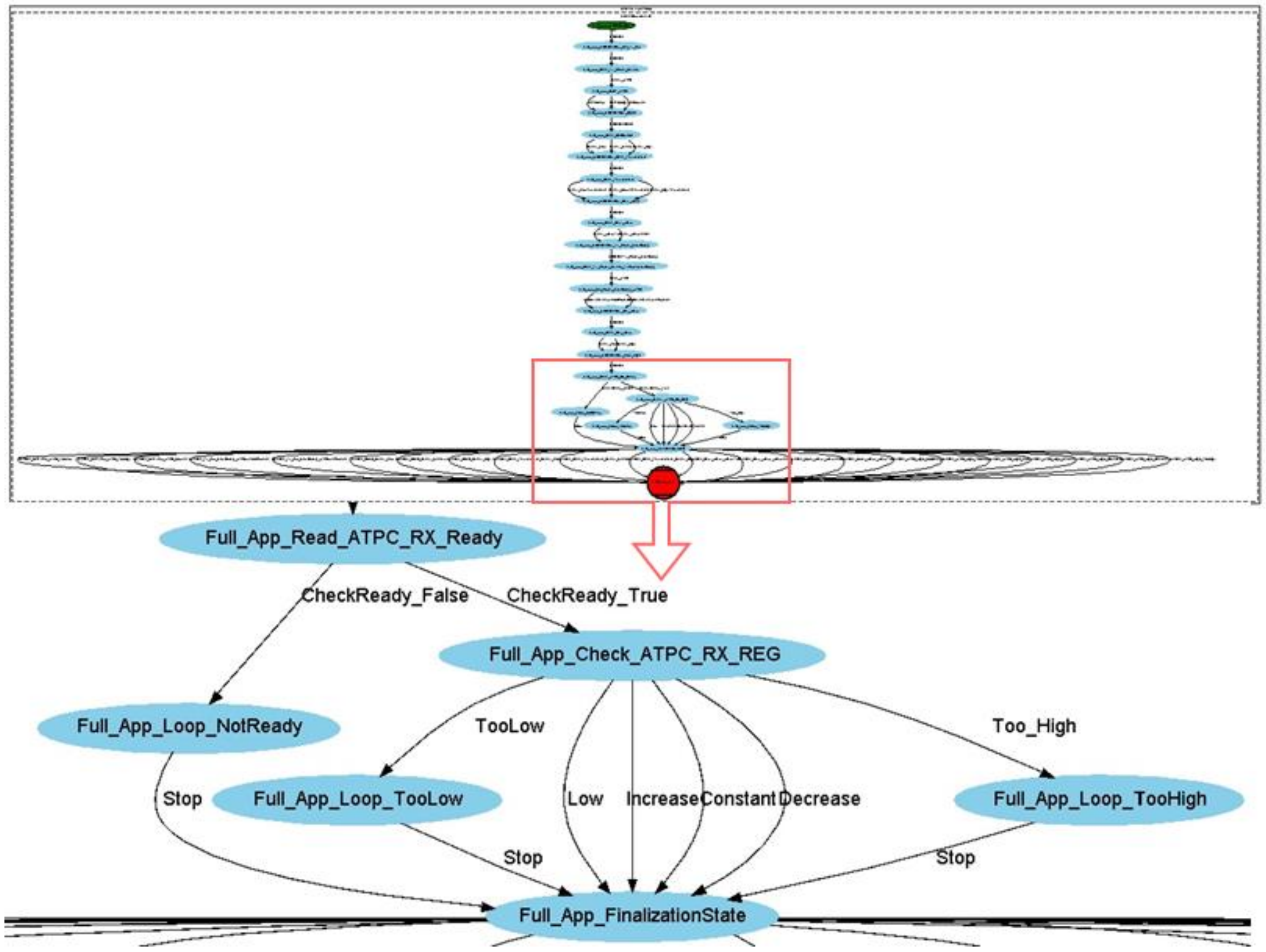
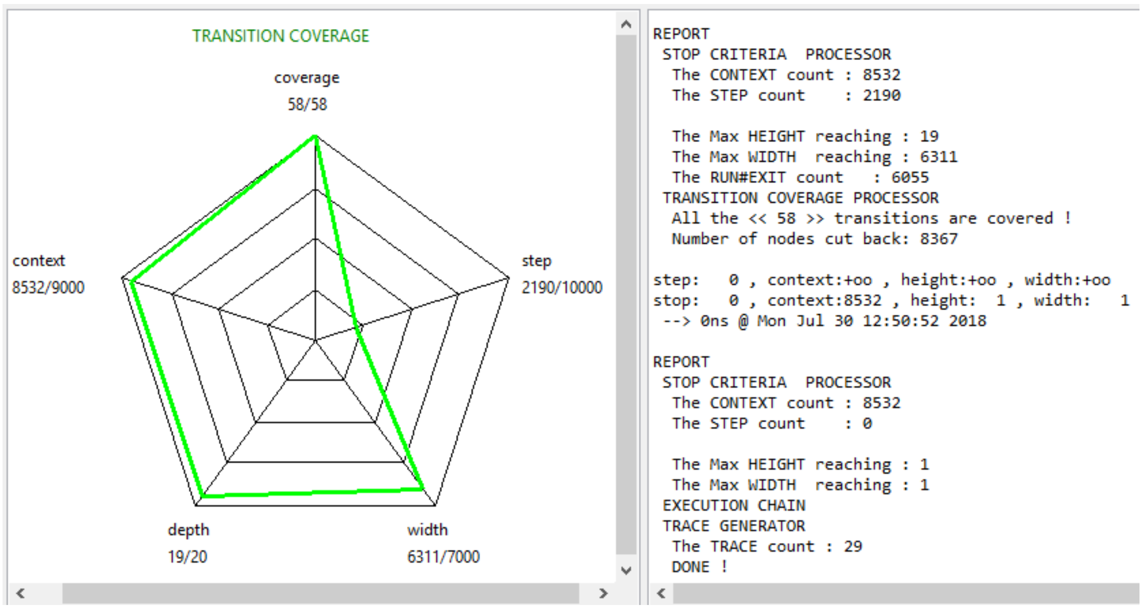


Figure 4-6 Telecom models for model validation

4.3.2 Performance estimation

In Performance Estimation, we have identified the composition patterns and defined the estimation model for each performance property related to the composition pattern, and we have developed an MBT component, i.e., model-based estimator, to take as inputs of the application's component network description and previous testing results to produce performance estimation results. Figure 4-8 shows the implementation details to achieve the model validation.



```
REPORT
 STOP CRITERIA  PROCESSOR
  The CONTEXT count : 8532
  The STEP count    : 2190

  The Max HEIGHT reaching : 19
  The Max WIDTH  reaching : 6311
  The RUN#EXIT count    : 6055
 TRANSITION COVERAGE PROCESSOR
  All the << 58 >> transitions are covered !
  Number of nodes cut back: 8367

step:   0 , context:+oo , height:+oo , width:+oo
stop:   0 , context:8532 , height:  1 , width:   1
 --> 0ns @ Mon Jul 30 12:50:52 2018

REPORT
 STOP CRITERIA  PROCESSOR
  The CONTEXT count : 8532
  The STEP count    : 0

  The Max HEIGHT reaching : 1
  The Max WIDTH  reaching : 1
 EXECUTION CHAIN
 TRACE GENERATOR
  The TRACE count : 29
  DONE !
```

Figure 4-7 Model simulation results for Telecom use case

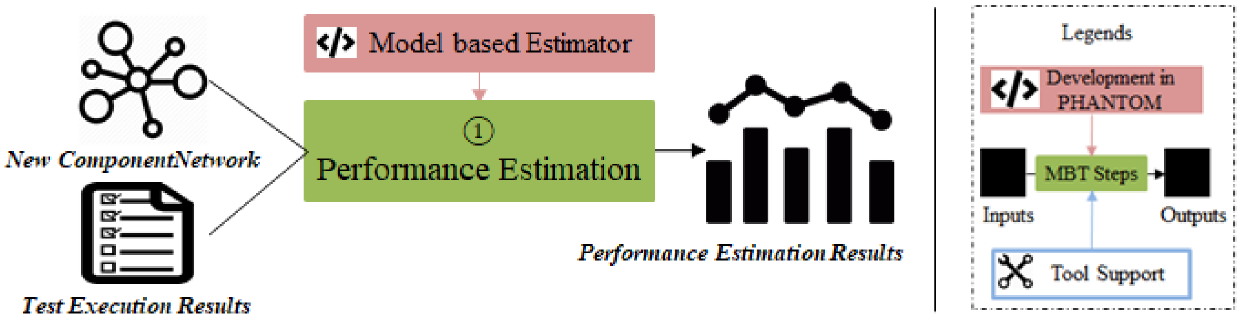


Figure 4-8 Performance estimation implementation

Model based Estimator. We have identified four types of composition patterns (i.e., sequence, parallel, condition and iteration) to represent the composition relations among application components. They cover most of the structures specified by workflow languages or workflow patterns [11]. For each performance property, we have defined an estimation model as shown in Table 4-1 to calculate the application's performance based on their individual components' performance and composition patterns.

Table 4-1 Estimation model for performance properties. k is the iteration time.

Property Name	Estimation Methods for Composition Patterns			
	Sequence	Parallel	Condition	Iteration

Execution Time (ET)	$\sum_{i=1}^{n} et_i$	$\max(et_i)$	$\max(et_i)$	$(et)*k$
RAM Usage (RU)	$\sum_{i=1}^{n} ru_i$	$\sum_{i=1}^{n} ru_i$	$\max(ru_i)$	$(ru)*k$
Reliability (RE)	$\prod_{i=1}^{n} re_i$	$\prod_{i=1}^{n} re_i$	$\min(re_i)$	$(re)^k$
Energy Consumption (EC)	$\sum_{i=1}^{n} ec_i$	$\sum_{i=1}^{n} ec_i$	$\max(ec_i)$	$(ec)*k$

Based on the estimation model, we have developed the model-based estimator. When given a component network description of a newly designed application, the model-based estimator analyses each component in the network and checks the previous testing results for each component and produces an estimated result for the newly designed application on each performance property. Figure 4-9 presents the performance estimation result of execution time for an HPC application. This application reuses three components A_1, A_2, A_3, and the component network is defined in Simulation5.

```
<estimation_phantom name="Estimation_HPCSimulation5" xsi:noNamespaceSchemaLocation="./estimation_phantom.xsd">
    <component_network>
        <application name="Simulation5" value="HPC">
    </component_network>
    <estimation_results>
        <execution_time value="8721ms">
    </estimation_results>
    <deployment>
        <mapping name="component_A_1" xsi:type="processing" persistency="full">
            <component name="A_1" />
            <processor name="P1" cpu-name="CPU1"/>
        </mapping>
        <mapping name="component_A_2" xsi:type="processing" persistency="full">
            <component name="A_2" />
            <processor name="P3" gpu-name="CPU1"/>
        </mapping>
        <mapping name="component_A_3" xsi:type="processing" persistency="full">
            <component name="A_3" />
            <processor name="P4" gpu-name="GPU1"/>
        </mapping>
    </deployment>
</estimation_phantom>
```

Figure 4-9 Performance estimation results for HPC application

4.3.3 Functional testing and non-functional testing

We have developed a number of MBT components for the test execution phase to enable functional and non-functional testing. We have developed MBT models for each use case and generated test cases from MBT models by applying selection criteria. We have also developed TTCN-3 publisher and codecs/decodecs to improve and concretize the generated test cases with real testing data and the system adapters to enable test execution. The development of models and other testing components starts in parallel with development and preparation based on specifications, and once the application is ready to run, concrete test cases are executed to conduct both functional testing and non-functional testing. Figure 4-10 shows the implementation details to achieve the model validation.

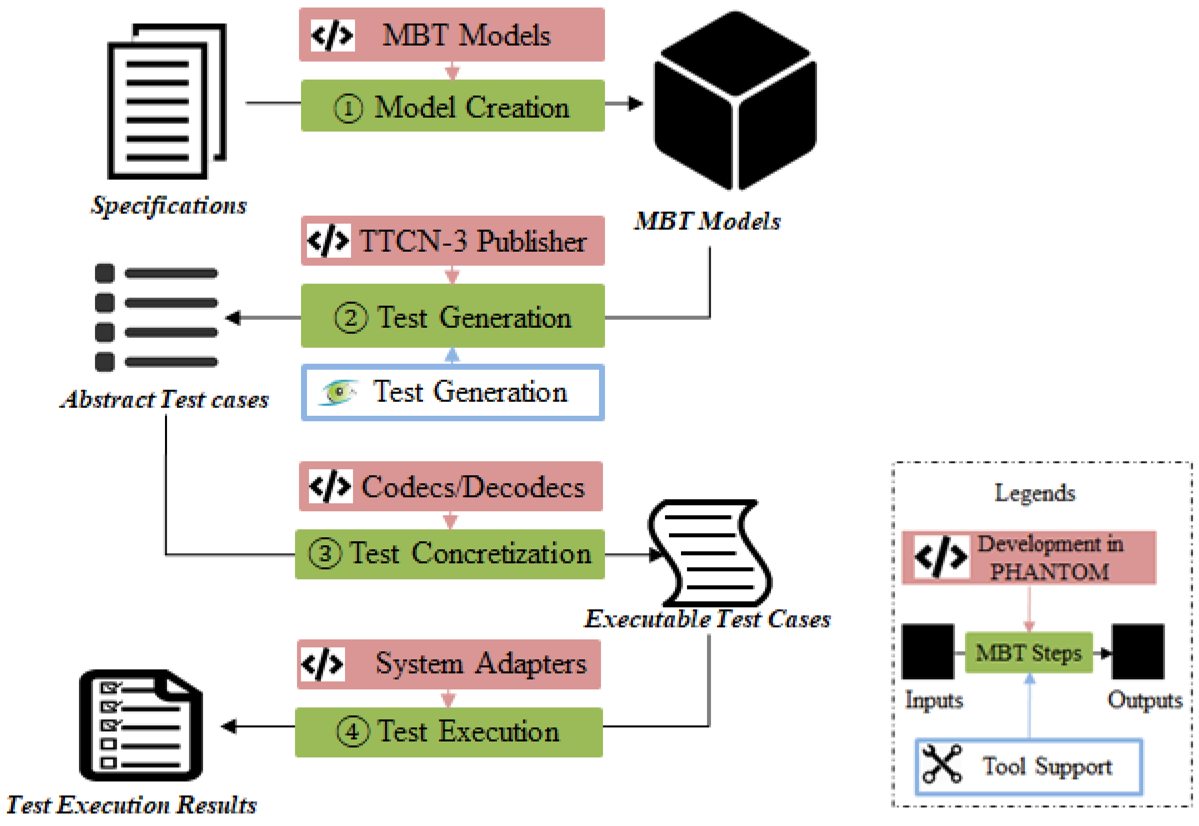


Figure 4-10 Functional and non-functional testing implementation

MBT models. Similar to the MBT models in early validation phase and following use case specifications, we have manually developed communicative state machine MBT models for each use case in xLIA. The MBT models are used to drive the test generation. As introduced in section 4.2, the MBT models for test generation focuses on global inputs and outputs relations and are created as independent models considering only application-level input and output while ignoring the inner communications amongst components. This consequently allows rather efficient test generation, execution and updates.

For each use case, the MBT models for test generation consist of two parts, i.e., functional testing model and non-functional testing model.

- Functional testing models represent all the dynamic behaviours of a SUT, and the corresponding generated test cases stimulates the SUT with specific inputs and collect SUT outputs to determine the correct functionality of SUT

- Non-functional testing models comprise representative SUT behaviours and the corresponding generated test cases stimulates the SUT with specific inputs and collect SUT performances to decide if the non-functional requirements are satisfied.

In PHANTOM, in order to facilitate test generation and management, we combined the functional and non-functional testing models as one model for each SUT. The following models for test generation are developed in PHANTOM:

- Telecom models for test generation
- HPC models for test generation
- Surveillance models for test generation

The Telecom use case application is function-driven in which a number of control flows dominate the application and the MBT models contain relatively rich states and transitions. The Surveillance and HPC use cases are data-driven in which applications are enabled by data flow to take inputs and generate outputs and thus the MBT models for the two use cases are rather simple on input and output data relations but focus on performance properties. The code below illustrates part of the surveillance model for test generation in xLia language.

```
@xlia< system , 1.0 >:
system<and> GMVApplicationSystem
{
    @property:
    … …
    @machine:
        statemachine< or > GMVApplicationMachine {
        @private:
        port input RequestUpload(InputImage);
        port output ResponseProcessedImage(OutputImage);
        port input RequestExecute(CommandLine);
        port output ResponseExecutionResult(CommandLineResult);

        @region:
        state<initial> State_ImageNotUploaded
        {
        transition UploadImage --> State_ImageUploaded
        {
           input RequestUpload(vInputImage);
        }}

        state State_ImageUploaded
        {
        /* Processing the image */
        transition ProcessImage --> State_ImageProcessed
        {
            input RequestExecute(vCommand);
            guard (GMV_APPLICATION == vCommand);
            output ResponseExecutionResult(NO_ERROR);
        }
        }
        … …
}
```

Test Generation Tool. The test generation is supported by tool to generate abstract test cases from MBT models following predefined selection criteria. In PHANTM, we use the DIVERSITY tool for test generation same as the model simulation tool.

Besides its open source nature and symbolic execution, another reason why we use DIVERSITY for test generation is that it generates test cases in the standard testing language TTCN-3 [12]. TTCN-3 is a standardized testing language developed by ETSI (European Telecommunication Standards Institute), with the advantages of multi-purpose testing support compared to other testing languages such as real-time support and distributed execution support, which highly align with the testing requirements and challenges in PHAMTOM distributed embedded environment.

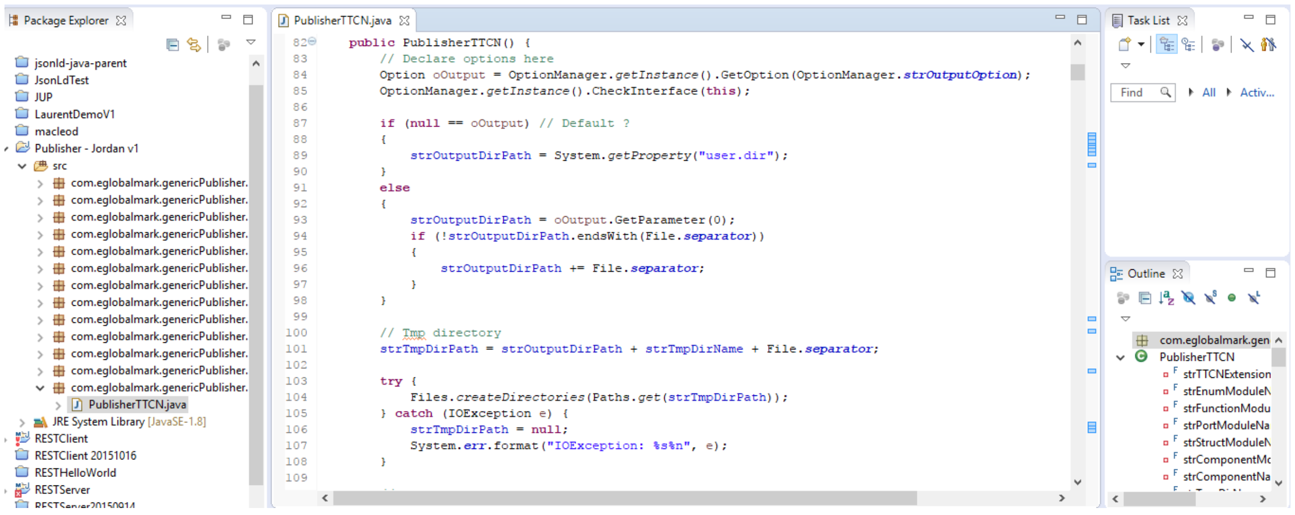
In PHANTOM, the MBT models are created in terms of communicative state machines with states and transitions, and we use the transition coverage as the selection criteria to generate test cases in order to cover all transitions and test all aspects in specifications. We have generated a suite of test cases for each use case application, and the code below illustrates part of the generated test cases for surveillance use case in TTCN-3.

```
module TTCN_TestsAndControl {

      import from TTCN_Declarations all;
      import from TTCN_Templates all;

      altstep RTDS_fail() runs on GMVApplicationSystem {
      }

      testcase TC_trace1() runs on runsOn_GMVApplicationSystem sys-
tem GMVApplicationSystem {
            activate(RTDS_fail());
            cEnv.send(RequestUpload_trace1_LINK_0)
            cEnv.send(RequestExecute_trace1_LINK_1)
            cEnv.receive(ResponseExecutionResult_trace1_LINK_2)
            cEnv.receive(ResponseProcessedImage_trace1_LINK_3)
            setverdict(pass)
      }
      … …
}
```

TTCN-3 Publisher. We have developed an MBT component, i.e., TTCN-3 Publisher, to further improve the test cases generated by DIVERSITY and provide additional testing function support.

As an open source tool for test generation, DIVERISTY is powerful for its generation algorithm and standard support, but also comes with some limitations (e.g., lack of documentation). Particularly, the TTCN-3 format that DIVERSITY uses doesn't fully align with the ETSI standard specification of TTCN-3, and the generated test cases contain errors. Thus, the primary role of TTCN-3 publisher is to assist the test generation process for error correction; furthermore, the generated test cases contain only basic testing functions to send and collect information, the TTCN-3 publisher further improves the generated test cases with better modularisation and timer functions to conclude a test case is inclusive when the execution time of a component is over than a threshold.

TTCN-3 publisher takes test cases generated from DIVERISTY as input and generates the updated TTCN-3 test cases with additional testing function support and error correction. Figure 4-11 presents part of TTCN-3 Publisher implementation in Java.



```
    public PublisherTTCN() {
        // Declare options here
        Option oOutput = OptionManager.getInstance().GetOption(OptionManager.strOutputOption);
        OptionManager.getInstance().CheckInterface(this);

        if (null == oOutput) // Default ?
        {
            strOutputDirPath = System.getProperty("user.dir");
        }
        else
        {
            strOutputDirPath = oOutput.GetParameter(0);
            if (!strOutputDirPath.endsWith(File.separator))
            {
                strOutputDirPath += File.separator;
            }
        }

        // Tmp directory
        strTmpDirPath = strOutputDirPath + strTmpDirName + File.separator;

        try {
            Files.createDirectories(Paths.get(strTmpDirPath));
        } catch (IOException e) {
            strTmpDirPath = null;
            System.err.format("IOException: %s%n", e);
        }

```

Figure 4-11 TTCN-3 Publisher Implementation

Codecs/Decodecs. Since automated test execution require test cases to be extremely precise, a test adaptation layer is required to provide the mapping between abstraction contained in the abstract test cases and the real data in the real implementations, to ensure the successful executions of the generated test cases. Codec/Decodec provides the function to transform abstract test cases to executable ones by providing mappings between the abstraction in MBT models and real testing data.

The Codec/Decodec is specific to MBT model and the corresponding test cases. The following three Codecs/Decodecs have been developed in TTCN-3 in PHANTOM to support the concretization of test cases for three use case applications.

- Codec/Decodec for Telecom application
- Codec/Decodec for HPC application
- Codec/Decodec for Surveillance application

The code below illustrates part of Codec/Decodec implementation for Surveillance application

```
module TTCN_Ports {

        import from TTCN_Enumerations all;
        import from TTCN_Structures all;

        import from GMV_Pixits all;
        import from ShellAdapter_Port all;
        import from ShellAdapter_PortTypes all;

        external function CheckReturnedImage(in charstring
p_ProcessedImage, in charstring p_CorrectProcessedImage) return
boolean;

        type port Generic_Port message {
                out RequestUpload;
```

```
             in ResponseProcessedImage;
             out RequestExecute;
             in ResponseExecutionResult;
       } with {
       extension
       "user ShellAdapter
       in(ShellAdapter_DownloadImage -> ResponseProcessedImage
       :function(f_dec_SADownloadImage_to_ResponseProcessedImage))
       out(RequestUpload -> ShellAdapter_UploadImage
       :function(f_enc_RequestExecute_to_SAUploadImage))
       in(ShellAdapter_ReturnValue -> ResponseExecutionResult      :
       function(f_dec_SAReturnValue_to_ResponseExecutionResult))
       out(RequestExecute -> ShellAdapter_Command
       :function(f_enc_RequestExecute_to_SACommand))"
       };
… …
}
```

System Adapters. System adapter provides communication channels to automatically execute test cases by connecting SUT with the test execution environment and data exchange. Generally, system adapter is specific to applications (i.e., Telecom, HPC, Surveillance) and running environments. In PHANTOM, we have developed individual system adapters for each use case interface as well as a universal system adapter for the PHANTOM Repository, which provides an abstraction over different applications and running environments, and thus a unified application-agnostic and environment-agnostic interface for users and testers.

The following system adapters have been developed in TTCN-3 and C++ to support the test execution for applications over different environments.

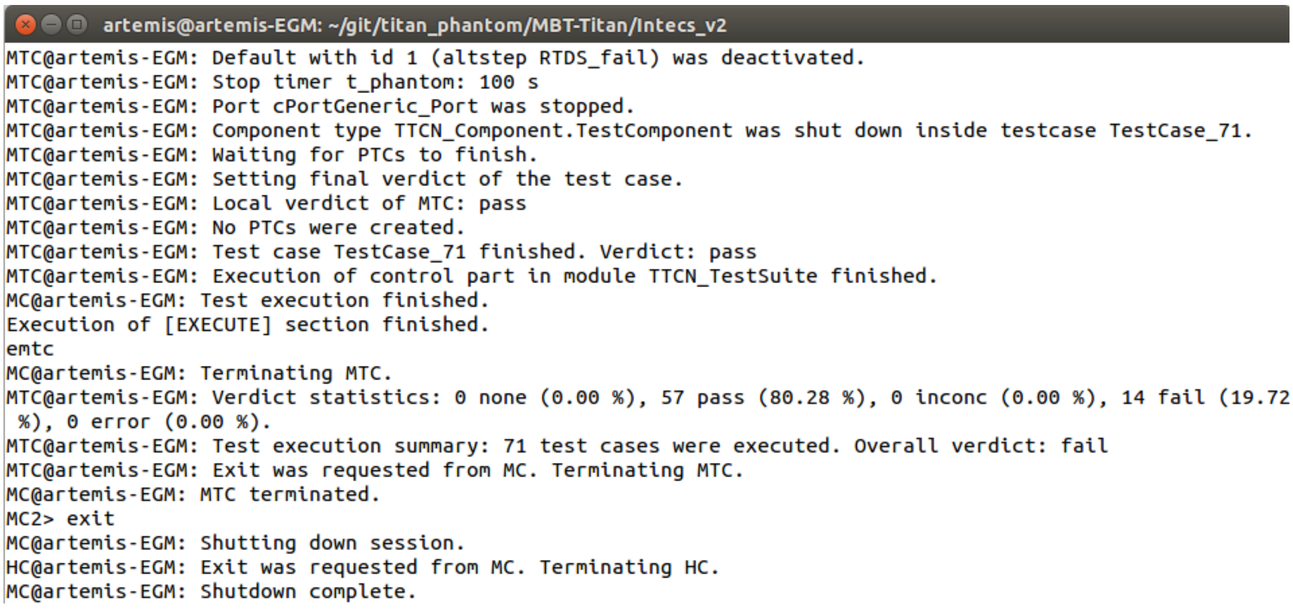
- System Adapter for Telecom Application over standard Linux
- System Adapter for Telecom Application over ZedBoard
- System Adapter for HPC Application over standard Linux
- System Adapter for Surveillance Application over standard Linux
- System Adapter for environment-agnostic PHANTOM Repository

Particularly, the latest efforts have been spent on developing the system adaptors for the newly integrated PHANTOM Repository so as to establish the data flow over PHANTOM Repository and enable the test execution over the PHANTOM environment-agnostic interface. The code below illustrates part of the System Adapter implementation for PHANTOM repository.

```
void ShellAdapter::send(const ShellAdap-
ter__PortTypes::ShellAdapter__Command& send_par, const COMPONENT&
destination_component)
{
     if(!is_started)
         TTCN_error("Sending a message on port %s, which is not
started.", port_name);
```

```
     if(!destination_component.is_bound())
         TTCN_error("Unbound component reference in the to clause of
send operation.");
    const TTCN_Logger::Severity log_sev = destina-
tion_component==SYSTEM_COMPREF?TTCN_Logger::PORTEVENT_MMSEND:TTCN_Lo
gger::PORTEVENT_MCSEND;
    if(TTCN_Logger::log_this_event(log_sev)) {
        TTCN_Logger::log_msgport_send(port_name, destina-
tion_component,
        TTCN_Logger::begin_event(log_sev, TRUE),
        TTCN_Logger::log_event_str("
@ShellAdapter_PortTypes.ShellAdapter_Command : "),
        send_par.log(),
        TTCN_Logger::end_event_log2str()));
}
    if (destination_component == SYSTEM_COMPREF) {
        void)get_default_destination();
        outgoing_send(send_par);
    }
    else {
        Text_Buf text_buf;
        prepare_message(text_buf,
"@ShellAdapter_PortTypes.ShellAdapter_Command");
        send_par.encode_text(text_buf);
        send_data(text_buf, destination_component);
    }
}
… …
```

The collection of the developed MBT components enables the execution of the generated test cases for each use case. In the PHANTOM implementation, functional testing and non-functional testing are conducted for all three use cases. Figure 4-12 presents the screenshot of the test execution for Telecom use case. The testing results are sent back to developers, and the use case applications have been updated correspondingly.


artemis@artemis-EGM: ~/git/titan_phantom/MBT-Titan/Intecs_v2

```
MTC@artemis-EGM: Default with id 1 (altstep RTDS_fail) was deactivated.
MTC@artemis-EGM: Stop timer t_phantom: 100 s
MTC@artemis-EGM: Port cPortGeneric_Port was stopped.
MTC@artemis-EGM: Component type TTCN_Component.TestComponent was shut down inside testcase TestCase_71.
MTC@artemis-EGM: Waiting for PTCs to finish.
MTC@artemis-EGM: Setting final verdict of the test case.
MTC@artemis-EGM: Local verdict of MTC: pass
MTC@artemis-EGM: No PTCs were created.
MTC@artemis-EGM: Test case TestCase_71 finished. Verdict: pass
MTC@artemis-EGM: Execution of control part in module TTCN_TestSuite finished.
MC@artemis-EGM: Test execution finished.
Execution of [EXECUTE] section finished.
emtc
MC@artemis-EGM: Terminating MTC.
MTC@artemis-EGM: Verdict statistics: 0 none (0.00 %), 57 pass (80.28 %), 0 inconc (0.00 %), 14 fail (19.72
 %), 0 error (0.00 %).
MTC@artemis-EGM: Test execution summary: 71 test cases were executed. Overall verdict: fail
MTC@artemis-EGM: Exit was requested from MC. Terminating MTC.
MC@artemis-EGM: MTC terminated.
MC2> exit
MC@artemis-EGM: Shutting down session.
HC@artemis-EGM: Exit was requested from MC. Terminating HC.
MC@artemis-EGM: Shutdown complete.
```

Figure 4-12 Test execution screenshot

4.3.4 Implementation summary

As a summary of the MBT implementation, we have developed all described components and conducted the four designed MBT activities for three use cases. Table 4-2 summarizes the MBT components developed in PHANTOM, in which some components are use case specific, some are interface specific, and some are use case independent. Table 4-3 illustrates the MBT activities conducted for each use case.

Table 4-2 MBT components developed in PHANTOM

	Use Cases		
	Telecom	**HPC**	**Surveillance**
MBT Components	*Models for Validation*	*Models for Validation*	*Models for Validation*
	Model based Estimator		
	Models for Test Generation	*Models for Test Generation*	*Models for Test Generation*
	TTCN-3 Publisher		
	Codec/Decodec	*Codec/Decodec*	*Codec/Decodec*
	System Adapter for Linux	*System Adapter for Linux*	*System Adapter for Linux*
	System Adapter for ZedBoard		
	System Adaptors for PHANTOM Repository		

Table 4-3 MBT Implementation in PHANTOM

MBT Activities	**Use Cases**		
	Telecom	***HPC***	***Surveillance***
Model Validation	√	√	√
Performance Estimation	√	√	√
Functional Testing	√	√	√
Non-Functional Testing	√	√	√

4.4 INTEGRATION ASPECTS

The MBT solutions has been entirely integrated into the PHANTOM platform, covering all PHANTOM stages with close interactions between MBT workflow and other PHANTOM components, so as to support of PHANTOM platform quality assurance and optimization.

The two following points are identified as the key objectives for integration, and the objectives have been specified and achieved in the integration implementation.

Objective 1: To define the interaction interfaces between the MBT workflow and the PHANTOM architecture components

During development, MBT takes design specifications from use cases and produces early validation results. During runtime, MBT stimulates SUTs in the PHANTOM environment and collect outputs files and performance information during the whole process. This involves the interaction between MBT components and use case applications, as well as MBT components and PHANTOM architecture components.

Objective 2: To specify the data flows over the interaction interfaces to enable the MBT testing activities

Use case developers use PHANTOM native interfaces to send inputs to application and get back output data; MBT testers reply on the same interfaces to collect files, send data, and get results. Particularly, when an application is deployed in the PHANTOM platform, MBT needs to test both the entire application and its individual components. Specific data flows need to be identified to enable the execution of either an application or only one individual component in PHANTOM.

4.4.1 PHANTOM platform interfaces

The PHANTOM platform mainly relies on the Repository to exchange information and enable control flow via notifications. The Repository provides HTTP interfaces and a publication / subscription mechanism to all PHANTOM components, which allows developers, testers and users to realize necessary data and control flow for task execution. The MBT relative information sent and received via the Repository includes:

- Use Case Specification documents, which describe the intended functionalities and non-functional requirements of applications.
- The Platform Description, Component Network, and Deployment Plan
- Input and Output data, which describe the necessary inputs for an application and its outputs at the end of execution;

Besides the Repository, the Application Manager is the second component to enable the interactions between users and testers to achieve the relative manipulation with application executions, the exchanged data include:

- Start Trigger, which notifies and starts the execution of an application
- End Signal, which notifies the end of execution when the execution of application is over.
- Performance Information, which describes the performance information related to a task.

4.4.2 MBT interaction flow with PHANTOM

The general interaction flow between MBT activities is to read the appropriate inputs (use case specification for Model Validation, component network for Performance Estimation), perform the validation or estimation, and then store the results back in the Repository. For functional and non-functional testing, the Application Manager is used to trigger the start of the deployed application/components. Once execution has completed, the Application Manager notifies the MBT tools which fetch the collected data.

1. Model Validation

Step 1. MBT gets the use case specification documents from Repository.

Step 2. MBT sends the model validation results to Reposiory after model validation.

2. Performance Estimation

Step 1. MBT gets the use case component network from Repository.

Step 2. MBT sends the estimated results to Repository after performance estimation.

3. Functional and non-functional testing

Step 1. MBT firstly gets from Repository a) platform description, and b) component network.

Step 2. MBT sends input data to the Repository.

Step 3. MBT sends deployment plan to the Repository. This step is optional, and it is only necessary when MBT tests individual component instead of the whole application: in this case, the deployment plan contains only description about one component and PHANTOM will use the new received deployment plan to deploy components to be executed.

Step 4. MBT sends the start trigger of execution to the Application Manager, and the platform starts the execution of application/components.

Step 5. Once the execution is over, the Application Manager sends to end signal to MBT.

Step 6. MBT gets the output data from the Repository. In case the output data is too large (e.g. a large image), the location where output data is stored will be sent back instead.

Step 7. MBT gets performance information from the Execution Manager. This step is necessary when non-functional testing is performed.

Step 8. MBT sends functional and non-functional testing results to the Repository.

4.5 INNOVATIONS BEYOND THE STATE-OF-THE-ART

4.5.1 Summary of new technologies/extensions developed

As illustrated in Table 4-2, 16 MBT components have been developed in PHANTOM to enable the four MBT activities of use case applications, including:

- **MBT Models for Validation**. Specific models in xLIA for Telecom, HPC and Surveillance applications. The MBT models we developed are based on communicative state machine, an extension of state machine, to consider parallel architecture and communication between components of PHANTOM applications.

- **Model based Estimator.** General MBT component in Java for all use cases to estimate performance of newly designed applications.

- **MBT Models for Test Generation**. Specific models for the three use cases including both functional testing and non-functional testing modelling aspects. The test generation is realized by symbolic execution for testing efficiency improvement.

- **TTCN-3 Publisher**. General MBT component in Java for all test cases. The TTCN-3 publisher provides error fixes, modulization and testing timers to better manage test complexity in PHANTOM.

- **Codecs/Decodecs**. Specific MBT components in TTCN-3 for three uses cases.

- **System Adapters.** Specific MBT components in TTCN-3 and C++ to support the test execution for all three use cases over PHANTOM environment, in Linux and another system adapter for Telecom use case on ZedBoard.

The MBT components developed in PHANTOM provide modularized functionalities and they are reusable for further testing activities. For examples, MBT models and test cases can be easily ported between different running environments; system adapters can be used to execute additional test cases in the same interfaces; TTCN-3 publisher can be applied to improve other TTCN-3 test cases.

Moreover, based on the design and implementation of MBT in PHANTOM, EGM has submitted an application of innovation patent.

- **MBT patent.** The MBT patent is prepared based the design and implementation of MBT activities in PHANTOM.

4.5.2 Full Prototypes functionality

The testing solution in PHANTOM is designed and implemented based on model based techniques and has advanced a step further for early testing enablement and testing effectiveness and efficiency improvement. Taking full advantages of the MBT innovations, MBT extensions have been made to parallel, embedded environments for component-based application testing. Concretely the full prototype provides the following four functionalities.

Model Validation. When given MBT models describing the dynamic behaviours of a SUT, model validation simulates the models with necessary data and detects the defects of deadlock or over designing. Based on the simulation results, a full MBT report is sent back to developers. Model validation is conducted based on the design specification of SUT and in parallel with the development process, so as to provide an efficient way to detect the defects in the design phase and prevent them from escalating to implementations.

Performance Estimation. Performance estimator is able to estimate the performance information about a newly designed component-based application. When given the component network description of an application, performance estimation analyses the composition relations among each component and deduce the performance of the whole application based on the previous testing results of each component. This is the second functionality, besides model validation, to provide early testing insights without running the real application in parallel with application development.

Functional Testing. Taking use case specifications as inputs, the functional testing is able to automatically generate test cases from MBT models, execute the test cases by simulating SUT with necessary inputs and collect corresponding outputs to check the correctness of functionality implementation. The generated test cases provide a systematic coverage to test system functionalities, and thanks to the MBT components developed in PHANTOM, the execution of test cases is automated to provide thorough execution results at the end.

Non-functional Testing. Non-functional testing takes use case non-functional requirements as testing focuses. The executed test cases do not cover all but only representative behaviours of SUT and collect execution related performance information to test if the all the non-functional requirements from specifications are satisfied. Besides the performance part, non-functional testing also helps developers to study the relation between parameter value and performance (e.g., in HPC use cases), which is an extension since the MBT does not only generate the test cases but the testing data as well.

5. PHANTOM MONITORING LIBRARY

The architecture of the PHANTOM run-time Monitoring Framework (MF) enables application optimization based on the understanding of both software non-functional properties and hardware quality attributes with regards to performance and energy consumption. Figure 5-1 shows the architecture of the MF, which is composed of an MF server, the MF client, and the MF library.

The MF-Library was integrated with the MF-Server under task 2.2 "Unified runtime monitoring implementation". However, we consider that MF-Library API, targeting to enable the collection of metrics at application level in a user-defined manner, has to be described in D3.2 because it is where are described the PHANTOM APIs ("Programming interfaces (APIs) per application class" task 3.2). This allows having all the tools used for the instrumentation of the applications in one place, as well as helps to provide a more clear view on the purpose and future use of the MF-Library API.

5.1 USE CASE REQUIREMENTS

The requirements that consist of two sets – the initial set of requirements, imposed by the use case providers at the beginning of the project and specified in Deliverable D1.1, and additional requirements that were obtained from the M18 evaluation – are provided in Appendix 5.

5.2 DESIGN SPECIFICATIONS

The PHANTOM Monitoring Library abstracts users from the metric collection process. The library provides for the users the mechanism to select the metrics to be collected during the execution of their applications at the system and application levels, as well as to configure the frequency of sampling. This mechanism is absent from existing tools.

The MF library additionally allows the collection of user-defined metrics. Each user metric is composed of a text label and a value or a set of values, which are automatically timestamped. This allows the users to measure, for example, the execution time of loops or subroutines by registering user-defined measures in the code. Such user-defined metrics are also missing from existing tools.

5.2.1 Architecture

The Monitoring Library has been developed as an extension to the monitoring solution elaborated by the EU projects EXCESS and DreamCloud. (see Figure 5-1).

The PHANTOM MF-Library (application-level monitoring and user-defined metrics collection) provides a user library and several APIs for instrumentation of the users' applications. It allows users to control and adjust the metric collection process by means of the user-level APIs. The PHANTOM Monitoring Framework follows the client-server architecture, according to which the runtime monitoring information is collected by means of a monitoring agent service (deployed on each of the monitored hardware resources) and transmitted to a centralised service (the monitoring server) that is usually deployed on a dedicated resource.

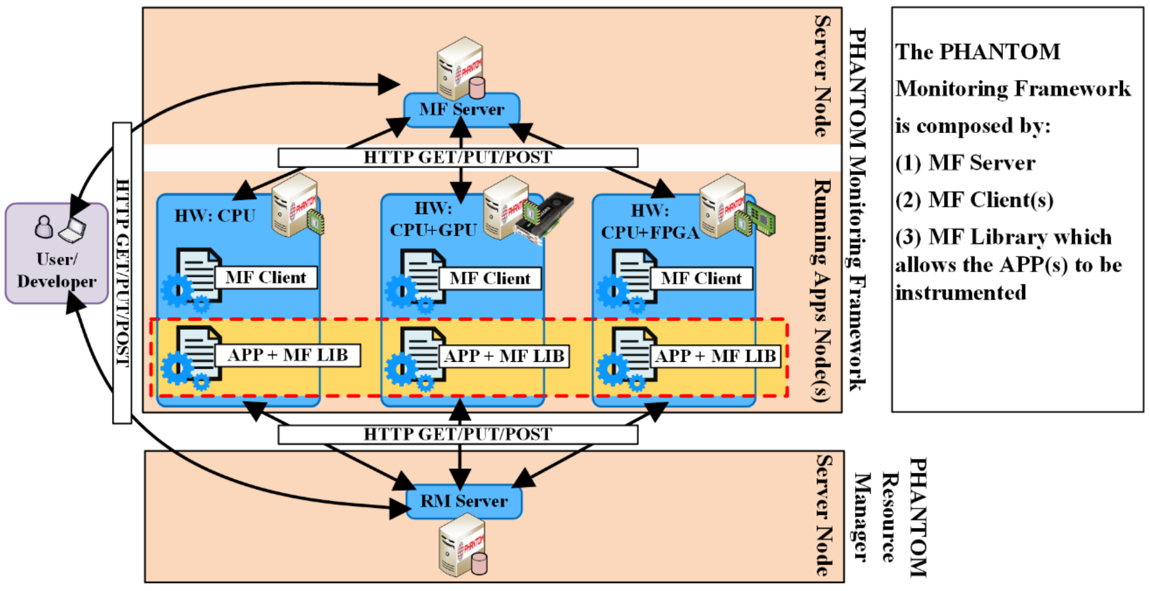


Figure 5-1 PHANTOM Monitoring Framework Architecture. In the figure is highlighted the instrumented applications with the MF library.

In order to achieve monitoring of metrics in both at System and Application Levels, the monitoring workflow is divided into two parts. As shown in Figure 5-2, devices can keep being monitored and alternatively the users can instrument their code and monitor at the application level using the MF Library, or use these two methods at the same time to obtain both kinds of collection of metrics.

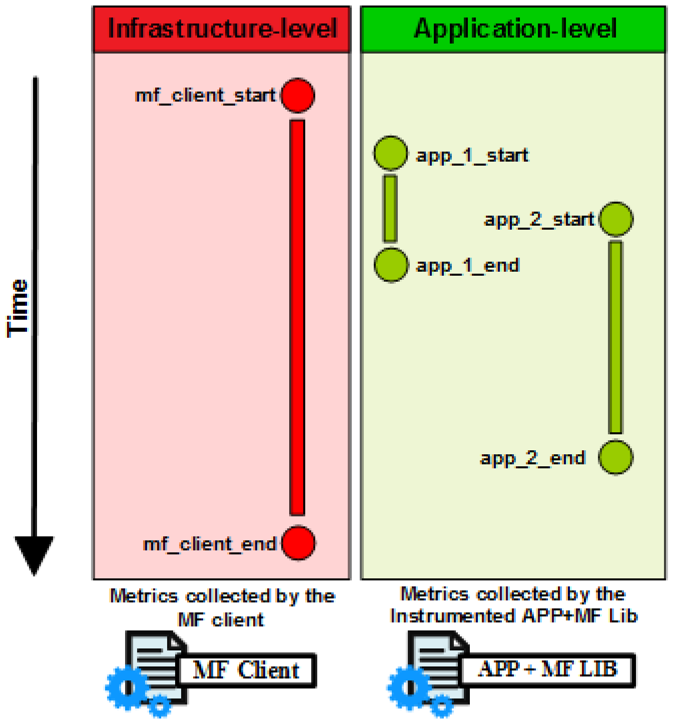


Figure 5-2 Infrastructure-level monitoring with the Monitoring Client, and Application-level monitoring with apps instrumented with the MF library.

Infrastructure-level, an MF client is firstly required to be installed on a specific target platform. Then the MF client collects the user-configured metrics at a customizable frequency and sends the collected data periodically to the server side.

Application-level metrics are obtained via a user library that allows code instrumentation and fine-grained monitoring. The Instrumented code can make use of the plugins supported by the Monitoring Client. In addition to the predefined application-level metrics, the user library supports user-defined metrics profiling as well.

Both levels of the monitoring-workflow are designed to be pluggable. Each plug-in has associated some particular metrics which are sampled and formatted periodically. The plugins to be loaded as the default configuration for each computation node are registered by the user at the Resource Manager Server.

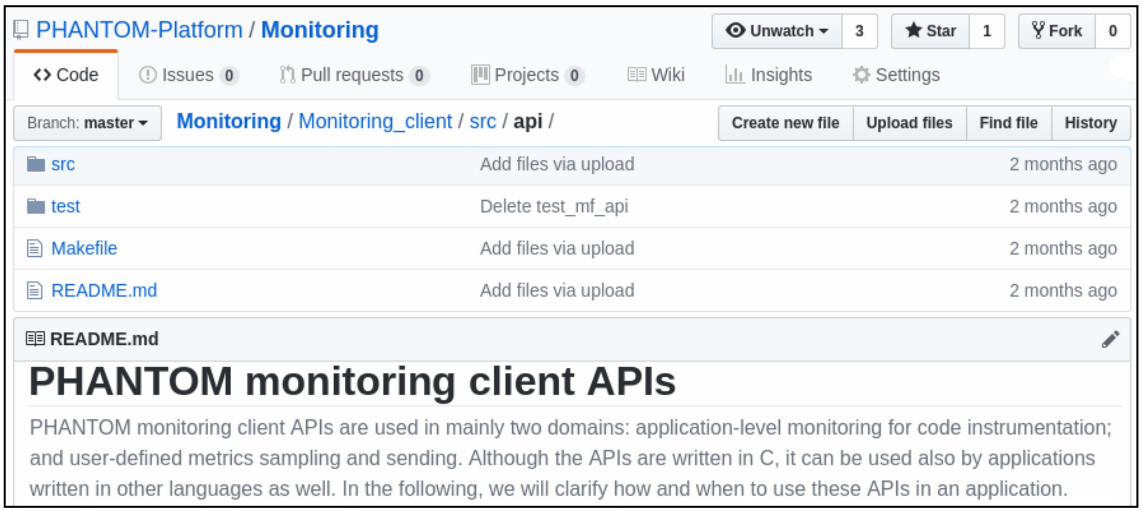
The monitoring services (at infrastructure and application levels) will use that configuration as a default one. However, the MF-Library provides to the users the flexibility to choose a different configuration for each application at the application level. Depending on the configuration of plug-ins and metrics, the plug-in manager checks the availabilities of the user-interested plug-ins, activates the associated metrics and prepares the sampling frequencies. Afterward, the thread handler creates for each activated plug-in a thread, which is responsible for sampling and publishing of the corresponding plug-in.

The application-level monitoring and user-defined metrics collection are both available in the API folder in the Repository, where there is the source code of the user library and several APIs for application code instrumentation. Some description for the interfaces and their usage introduction are given in Section 5.4 and the full details on the other components of the MF are provided in D4.2.

5.3 IMPLEMENTATION DETAILS

The PHANTOM MF-Library (application-level monitoring and user-defined metrics collection) provides a user library and several APIs for instrumentation of the users' applications. The implemented code (in C) of the user library provides a set of functions for running the Monitoring plugins developed for the MF-Client. In addition, those capabilities are extended with a set of functions for the collection of user defined metrics. The source code of the MF-Library is available on the folder */api* within the MF-Client source code at the GitHub . In that folder there is instructions and an example of an instrumented application. The Figure 5-3 show the view of the GitHub webpage, which publicly accessible at the link:

https://github.com/PHANTOM-Platform/Monitoring/tree/master/Monitoring_client/src/api



Monitoring / Monitoring_client / src / api /

src	Add files via upload	2 months ago
test	Delete test_mf_api	2 months ago
Makefile	Add files via upload	2 months ago
README.md	Add files via upload	2 months ago

README.md

PHANTOM monitoring client APIs

PHANTOM monitoring client APIs are used in mainly two domains: application-level monitoring for code instrumentation; and user-defined metrics sampling and sending. Although the APIs are written in C, it can be used also by applications written in other languages as well. In the following, we will clarify how and when to use these APIs in an application.

Figure 5-3 Screenshot of the /api folder in the GitHub repository

The implementation of each plug-in is collected in the folder /plugins.

The pluggable design of MF-Client and the MF-Library allows a dynamic configuration of their functionalities based on the users' configuration at run-time, but also easy its extension in the future when new metrics and hardware are to be monitored.

For details of each plug-in's implementation see the Appendix "*Monitoring Client's Plug-ins*" in D4.3, which will be extended in D4.4 with the description FPGA-power plug-in.

5.4 INTEGRATION ASPECTS

This section describes the required instrumentation of the user code in order to collect the metrics requested in the Initial Set of Requirements.

5.4.1 Application-level APIs

The client interfaces detailed below were developed and included in the first prototype release. In addition to collecting generic metrics of performance and power, the APIs are capable of monitoring application-specific metrics as well.

As a point of reference, the code below shows the normal order of calling the user-library functions. The code starts the monitoring of a set of metrics calling the function mf_start, later collects some user defined metrics, and finally, it requests the end of monitoring (mf_stop) and forward the remaining buffer of metrics (mf_send).

```
/*
Start monitoring of the predefined metrics for sub-components of an
application. Data are stored at first locally. Required input
parameters should include the MF server URL, the name of the
platform, where the application runs, and the metrics' name and
sampling frequency.
*/
char *mf_start(char *server, char *platform_id, metrics *m);

/*
```

```
Send user-defined metrics with given metric's name, value, and
current local timestamps to the PHANTOM MF server. Returns the
status of the operation completion.
*/
// thread_id  is used to set independent buffer for each thread,
// set thread_id as 0 if there is only a single thread
int mf_user_metric(char *metric_name, char *value, int thread_id);

/* or integer value metric*/
(metric_query *) my_query=new_metric("user_metric" , int thread_id);
my_query=add_int_field(my_query, "", 1, (int [])(20} );
submit_metric(my_query);

/*or string value metric*/
(metric_query *) my_query=new_metric("user_metric" , int thread_id);
my_query=add_string_field(my_query,      "",      1,      (char      *
[]){"newentry"} );
submit_metric(my_query);

/*or JSON string metric*/
char *my_t = mycurrenttime_str();
sprintf(string_a,"\"user_metric\":{\"
loops\":12,\"user_timestamp\":%s}", my_t);
submit_metric_json(user_string, int thread_id);

/*
Stop monitoring of the predefined metrics when the sub-component is
finished.
*/
void mf_end(void);

/*
Send locally-stored predefined metrics to the PHANTOM MF server. The
unique generated execution ID will be returned on success.
*/
char    *mf_send(char    *server,    char    *application_id,    char
*component_id);
```

For more complex and detailed examples, see *Appendix 6. Examples of Registering User Defined metrics.*

5.5 INNOVATIONS BEYOND THE STATE-OF-THE-ART

The following major innovations are identified for the Monitoring Library:

- **Management of monitoring configurations on heterogeneous systems.** The PHANTOM Monitoring Framework allows an administrator to register the description of the hardware components of the compute nodes, as well as their default configuration for monitoring. This task would normally be carried out only when new nodes are added to the system or when the existing nodes are updated. In this way, the use of the framework is eased to the end users, since it frees the users of the configuration task as well as the necessary expertise to do it.

- **Monitoring of user-defined metrics for the application-level monitoring.** Without the need of installing any additional monitoring tools, application (or user) specific metrics can be monitored. For this purpose, the Monitoring Library offers the users a set of light-weight, hardware-agnostic APIs for injecting the instrumentation into the application code. This provides a tight integration of the Monitoring Framework with the user application.

- **Highly customizable monitoring settings.** The PHANTOM Monitoring Framework was designed with the user-friendliness in mind – the users can specify the metrics to be automatically collected, the flush time interval, the configuration of collected metrics, etc., independently for each application, which is not the case for the other, alternative approaches.

This allows defining a more relaxed sampling frequency for monitoring at the infrastructure level (when may not be an application running on the node), and a more exhaustive frequency at the application level for those metrics that the user is interested in. In particular, the user can define different sample rates for each plugin. In particular, the sampling frequency can be modified during the execution of the application (by modifying the configuration parameters).

5.5.1 Background technologies utilised in the development

Among a broad set of available open-source tools dealing with infrastructure monitoring (e.g. Zabbix, Nagios), application-level optimization (e.g. Paraver, Vampir), we were unable to identify any technology that would fulfill the user's requirement to the Monitoring Library – i.e. allow the collection and centralised processing of the application-specific data.

Therefore, the user's extensions of the Monitoring Framework – the Monitoring Library – were largely developed from scratch (with the reuse of the design outcomes of the EXCESS project [13]). The elaborated API syntax follows the one used in the well-established tools for parallel applications profiling like Extrae [14] and Vampir-Trace [15] and is tightly integrated with the Monitoring Client functionality (cf. D4.2).

5.5.2 Summary of new technologies/extensions developed

The PHANTOM Monitoring Library provides a way to monitor user-defined metrics **to** analyze the changes in the collected metrics across multiple executions of their application. Also for the MBT tools, the user-defined metrics can give important hints on the properties that should be considered during modeling.

After the execution, users can perform analytics on their customized metrics in the same way as they would do for any default metric, using the macro- or micro-querying functionality of the Monitoring Server (see in D4.2).

The Monitoring Library also provides a very flexible way to configure all default metrics that should be collected for the application by the Monitoring Client. For this purpose, the function *mf_start* was extended to accept the list of plug-ins that need to be activated for each specific application run.

5.5.3 Source release and GIT repositories

The source code of PHANTOM Monitoring Framework is released via GitHub. The source code of the MF-Library is available online with the source code of the Monitoring Client because of both share source code and multiple libraries. The code can be accessed via the following link:

https://github.com/PHANTOM-Platform/Monitoring/tree/master/Monitoring_client

6. CONCLUSION

This deliverable reports the final development of the tools and technologies that will support the activities of three modules of PHANTOM in WP3. Those components are Parallelization Toolset, Programming Interface, Model Based Testing and the Monitoring Library[1]. For each module are identified:

- the requirements from the use cases that each module must fulfil and preliminary feedback from M18.
- the design aspects and decisions taken during specification;
- the implementation details and technologies used in the implementation of those tools;
- how the developed tools will communicate with the other PHANTOM modules;
- the summary of innovations and full prototype functionality.

Regarding to the development of the Parallelisation Toolset, 15 requirements were identified from the use cases to be taken into consideration. Design of two functionalities were discussed: Code Analysis, responsible for the identification of parallelisable code inside components of a PHANTOM application; and Technique Selection, responsible for the selection of the most appropriate component versions (e.g. OpenMP, CUDA, FPGA etc.) that allows a more efficient implementation of the deployment plan and injection of the API/annotations to each component's source code. It is also described how components will be implemented to run on FPGA-coupled devices, having identified the APIs and directives that need to be respected.

In the case of the Programming Interfaces, the use cases provided 12 requirements. To meet these requirements, 3 groups of APIs were described: Shared API, to handle shared memory; Queue API, allowing the usage of blocking FIFO data items on distributed memories; and Signal API for coordinating the execution of components. Due to the needs of the use cases, the Mutex protocol has also been developed, providing code block mutual exclusion between the components. Moreover, an API for specifying CPU-GPU communication was also described.

Model Based Testing (MBT) addresses 3 requirements from the use cases. Use cases were studied in detail in order to allow the understanding of the technical challenges that they provide to the execution of Model Based Testing methodologies. MBT components have been developed in PHANTOM and the MBT workflow has been achieved to deliver end-to-end model validation, performance estimation, functional testing and non-functional testing for all three use case applications; MBT results have been sent back to developers and the applications has been updated correspondingly through iteration cycles between MBT and application development. The MBT integration in PHANTOM was also defined and developed according to the integration objectives. The innovation of MBT mainly relies on enabling early functional and non-

[1] The MF-Library API was integrated with the MF-Server under task 2.2 "Unified runtime monitoring implementation". However, it is described in this deliverable with the purpose of having all the tools used for the instrumentation of the applications in one place.

functional testing, improving testing effectiveness and efficiency, and on the MBT extension to component-based applications in parallel and embedded environments.

The requirements identified for Monitoring Library were 12 from the use cases, and 3 additional from the questionnaires filled by the Use Case Partners. The Monitoring Library provides the run-time Monitoring Framework with capability to capture non-functional properties and hardware quality attributes with regards to performance and energy consumption aspects. The PHANTOM Monitoring Library abstracts users from the metric collection process, allowing the focus on application optimization. The library provides for the users the mechanism to select the metrics to be collected during the execution of their applications at the system and application levels, as well as to configure the frequency of sampling.

REFERENCES

[1] ROSE Compiler, http://rosecompiler.org/

[2] autoPar, http://rosecompiler.org/ROSE_HTML_Reference/auto_par.html

[3] LLVM, https://llvm.org

[4] Clang compiler, http://clang.llvm.org

[5] Vivado High-Level Synthesis, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[6] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada, June 2000.

[7] Kramer, A., Legeard, B.: Model-based testing essentials: guide to the ISTQB certified model-based tester foundation level. John Wiley & Sons Inc, Hoboken, New Jersey (2016)

[8] Thomas A. Henzinger and Joseph Sifakis. 2006. The embedded systems design challenge. In Proceedings of the 14th international conference on Formal Methods (FM'06), Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-15. DOI=http://dx.doi.org/10.1007/11813040_1

[9] Model-Based Testing: An Approach with SDL/RTDS and DIVERSITY

[10] Eclipse Formal Modelling Project, https://projects.eclipse.org/projects/modeling.efm

[11] Moscato, F., Mazzocca, N., Vittorini, V., Di Lorenzo, G., Mosca, P., and Magaldi, M. Workflow pattern analysis in web services orchestration: the BPEL4WS example. Proceedings of the First international conference on High Performance Computing and Communications, Springer-Verlag (2005), 395 – 400.

[12] TTCN-3, http://www.ttcn-3.org/

[13] EXCESS, http://www.excess-project.eu/

[14] Extrae, https://tools.bsc.es/extrae

[15] VAMPIRTRACE, https://tu-dresden.de/zih/forschung/projekte/vampirtrace

[16] Palabos, http://www.palabos.org/

Appendix 1. Initial Requirements of Parallelization Toolset

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U3	Parallelization of sequential application code, when complemented by parallelization instructions provided by the user	SHALL	YES
U4	Automatic identification and parallelization of regions of sequential application code	SHOULD	YES

Parallelization of sequential code is performed by Code Analysis that identifies the parallelizable regions in the code and inserts the necessary OpenMP annotations that allows the application to use multiple treads for its execution. User instructions are enabled by the PHANTOM Programming Model to enhance the tool's understanding of the code, improving the analysis and producing better results.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U5	Support for multi-threaded concurrent tasks, including communication and synchronization	SHALL	YES

The tool parallelizes each component using the OpenMP library enabling its multi-threaded capabilities. The components' execution is also implemented as a set of multi-threaded tasks that are running concurrently and exchange data using communication standards like POSIX and MPI.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U6	Support of parallelization, influenced by non-functional requirements information	SHOULD	YES

Parallelization Toolset creates more applicable versions of the parallelized components to provide the necessary flexibility to the Multi-Objective Mapper that is responsible to provide an optimized mapping for the satisfaction of non-functional requirements.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U7	Support for communications data-centric applications (e.g. automatic scaling of components to the actual size of data to be processed)	SHALL	YES

A suitable interface (as discussed in the following paragraphs) is provided for the support of data-centric applications, managing, if necessary, different scales of the data or even slicing them into smaller components according to the application's specifications.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U8	Support for component-based application design	SHALL	YES

The Parallelization Toolset uses the Component Network provided by the user to assume a collection of components interacting with each other while running in parallel.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U14	Exploitation of SIMD instructions sets provided by CPUs	SHOULD	YES

SIMD exploitation is supported by the insertion of specific OpenMP annotations (e.g. reduction) that use the CPU's extended capabilities for optimized execution. The annotations are injected in the code during the generation of the OpenMP versions of the components as described in the following paragraphs.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U19	Generation of target dependent parallel code for all mandatory target platforms without user involvement when sufficient annotations are provided.	SHOULD	YES

The Parallelization Toolset provides support for CPU, GPU and FPGA deployment. The tool successfully transforms the code in order to provide the necessary interface to external devices and provide those versions alongside the ones transformed to run in parallel on CPU-only, multithreaded systems.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U21	Automation of transferring data to/from different memories according to the component data model	SHALL	YES

This functionality is currently assigned to the Deployment Manager and the PHANTOM Programming Interface who work in synergy with the PT and are responsible to implement any data transfers between the components optimally.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U22	Support for indication of application blocks to be parallelized	SHALL	YES

The Programming Model enables the user to indicate specific regions of code that don't face certain dependencies, assessing the Parallelization Toolset to exclude those dependencies from its analysis unlocking in this way more parallelization capabilities for the component.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U23	Support for indication of data dependencies, defining how data can be partitioned/split among the parallel application components	SHALL	YES

Component replication is supported by the Parallelization Toolset allowing the user to include the necessary annotations in their code to indicate the components that are able to be replicated, splitting the data to each copy of the component.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U32	Support for application source code developed in C	SHALL	YES
U33	Support for higher level language such as Java and C++	MAY	YES

The Parallelization Toolset provides support for a lower level programming language such as C, as well as a higher level language such as C++, giving the user a basic flexibility on the level of usage they want to apply on the framework.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U37	Support for exposing the generated parallel code to the user	SHALL	YES

The parallel code generated by the PT is being exposed to the user through a well-viewed panel provided by the tool's GUI. All the modified versions as well as the IPCores generated by IPCore generator are stored in the Repository and can be accessed by the user.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U38	User modifications of the generated parallel code subject to restrictions or protected segments	SHOULD	NO

During development no such requirement was needed to be implemented based on the use cases applications, hence it was considered unnecessary.

Appendix 2. INITIAL REQUIREMENTS OF PROGRAMMING INTERFACE

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U5	Support for multi-threaded concurrent tasks, including communication and synchronization	SHALL	YES

Suitable APIs (Programming Interface) are provided to support both communication and synchronization between concurrent tasks.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U7	Support for communications data-centric applications (e.g. automatic scaling of components to the actual size of data to be processed)	SHALL	YES

A suitable interface (as discussed in the following paragraphs) is provided for the support of data-centric applications, managing, if necessary, different scales of the data or even slicing them into smaller components according to the application’s specifications.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U8	Support for component-based application design	SHALL	YES

The implementation of the Programming Interface is designed to guarantee sufficient communication and synchronization between different components that constitute a whole application.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U19	Generation of target dependent parallel code for all mandatory target platforms without user involvement when sufficient annotations are provided.	SHALL	YES

OpenMP annotations are added by the PT in the sequential code transforming it into multithreaded code for CPU targets. Regarding GPU and FPGA targets, an API for GPUs is supported providing the necessary interface for executing code on the corresponding targets, while the IP Core Generator module is responsible to provide auto-generated IP Cores for execution on FPGAs.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U20	Provision of constructs or abstractions to deal with non-uniform and uniform memory, hiding the underlying data transfer details	SHALL	YES
U21	Automation of the process of transferring data to/from different memories according to the component data model	SHALL	YES

The Programming Interface, with the help of the component network, provides communication between the different components by hiding at the same time the details of the implementation concerning any type of memory transfers.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U24	Provision of means for the developer to describe the composition of hardware components and interactions for the target platform	SHALL	YES
U28	Provision of a data model for specification of input and output data	SHALL	YES

The component network as well as the platform description are designed to allow the user to provide specifications about data transfers and the hardware components that are available.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U32	Support for application source code developed in C	SHALL	YES
U33	Support for higher level language such as Java and C++	MAY	YES

The Programming Interface provides support for a lower level programming language such as C, as well as a higher level language such as C++ giving the user a basic flexibility on the level of usage they want to apply on the framework.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U47	Support for Telecom specific application classes where domain-specific libraries are commonly utilized	SHALL	YES

No constraint is defined in the Programming Model that can exclude domain-specific libraries. Any such library is supported by PHANTOM as long as it is included by the user in the code.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U86	Support for application specific communication bus/protocols	MAY	YES

The Programming Model can be used side by side with the default C/C++ environment, which means that any external communication protocol/library can be used independently to the PHANTOM protocols.

Appendix 3. INITIAL REQUIREMENTS OF MODEL BASED TESTING

The initial use case requirements that MBT addresses in PHANTOM project is introduced in the table below.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U29	PHANTOM should provide a means to test the correct functioning of the application when it is mapped onto heterogeneous HW targets	SHOULD	YES
U30	PHANTOM should provide mechanism to test the correct APIs implementation	SHOULD	YES
U31	PHANTOM should provide an API for implementation of tests (similar to JUnit for Java)	SHOULD	YES

The four MBT activities in two phases ensure the correct functioning of applications (U29) and APIs (U30), but also provide testing results for application performance to check if the non-functional requirements are met. The API for test implementation (U31) consists of two parts: 1) the API for model validation, functional and non-functional testing is based on xLIA language (eXecutable Language for Interaction & Assemblage) [9]. Users can create their own MBT models for validation and test generation, and the generated test cases can be executed by use of the developed MBT components; 2) The API for performance estimation is the same as PHANTOM component network. Users can create their own application component networks and run the performance estimation to get estimation results.

Appendix 4. TESTING REQUIREMENTS OF USE CASES

In addition to the general requirements on PHANTOM platform, specific use case requirements related testing are also identified by each use case in terms of specification documents. The specifications describe the expected functions and performance of each use case, which are served as the basis for testing activities. Each of the three use cases has identified several functional aspects (input, output, precondition and post condition) to test for the entire application as well as their components. The latest specifications are briefly introduced as follows.

Surveillance specification

GMV develops and markets a surveillance system to provide added-value support to maritime situational awareness via Earth Observation technologies. It is a fully automatic and modular tool that permits detecting and categorising ships by combining the information inferred from Synthetic Aperture Radar data with transponder-based polls (such as the Automatic Identification System). The information from these different sources is integrated and provided, as a service, through an advanced GeoPortal web interface.

The updated surveillance application defined using PHANTOM programming model is named Ship Detection and has a number of components as illustrated in Figure A- 1; the way how components are composed is also illustrated in the same figure. The functional specification of each component is summarized in Table A- 1.

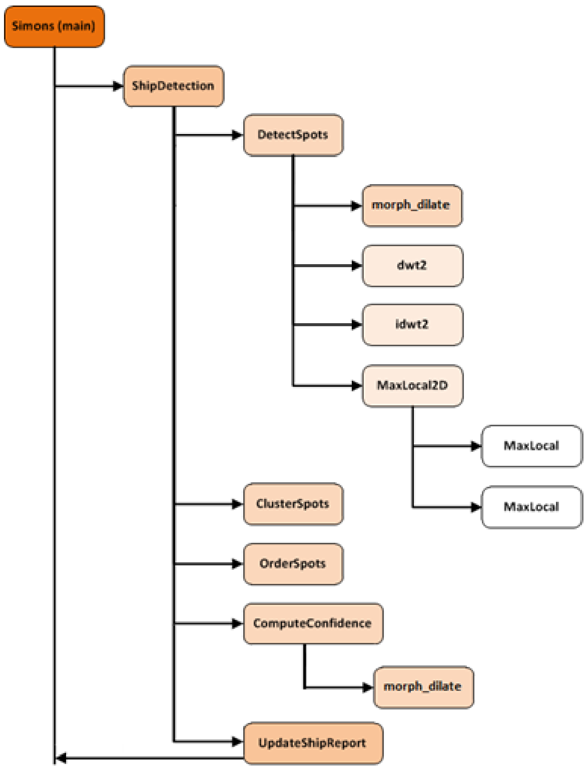


Figure A- 1 Surveillance use case components

Table A- 1 Functional specification of Surveillance Use Case

Application/ Component	Inputs	Outputs
ShipDetection	- block_id - imageBlock - imageBounds - coastlineSections - Att: extracted metadata from image - miscellaneous	- shipReport
morph_dilate	- block_id, mat and mask	-dilated
DetectSpots	- block_id - coastlineSections - image, maskLand, misc	- SpotsT - SpotsD
dwt2	- block_id - sig, nm, J	- ca: approximation coefficients matrix - ch: horizontal coefficients matrix - cv: vertical coefficients matrix - cd: diagonal coefficients matrix
idwt2	- block_id - isg, nm, J	- dwt_inv: approx. and coefficients matrix
MaxLocal2D	- block_id - image_ship, max_spots, type	- values: local maximum - row_ind: Row indexes - column_ind: Column indexes
MaxLocal	- block_id - input_data - dim: matrix dimension to compute	- max_ind: array of the max indices
ClusterSpots	- block_id - ShipReportPre, image, spots - valuesPre - LoopIndex: index for storage decisions - detection: constant configuration	- SpotsTemp: clustering for input spots
ComputeConfidence	- block_id - spots, shipReport, image, detection - imageBounds, misc, maskLand	- shipReport
OrderSpots	- block_id - spots, shipReport, shipReportPre - valuesPre, LoopIndex, soptsTemp - valuesClust	- spots - shipReport
UpdateShipReport	- block_id - shipReport, att, misc	- shipReportGlobal - indReport

Besides the functional aspect, the non-functional requirements for surveillance requires that the execution time of application should be less than 1600 seconds.

Telecom specification

Intecs develops a system for Automatic Transmission Power Control (ATPC). ATPC refers to a functionality supported by the high frequency radio circuits (rf) that allows to control the power of the transmitted signal based on the received signal level on the remote end antenna exchange via radio embedded ATPC protocol.

The ATPC application architecture is decomposed in a set of "atomic" components each one activated at regular and specific time interval (polling time). Each component runs autonomously on the base of the information from hardware or from shared areas as illustrated in Figure A- 2.

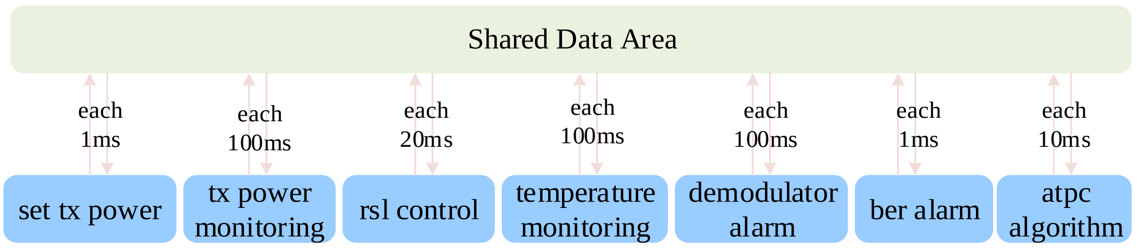


Figure A- 2 Telecom Use Case Architecture

Table A- 2 identifies each component, the polling interval and provides a brief description of its activity, highlighting the relevant input, output data (additional shared data are detailed in the test cases, e.g. used for compensation, alarms, etc.) and alarms.

Table A- 2 Functional specification of Telecom Use Case

Component	Poll (ms)	Input	Output	Alarms
set tx power	1	txpwr_dac_val	DAC(1)	
tx power monitoring	100	ADC(3)	detected_tx_pwr	SQUELCH TX_PWR
received signal level control	20	ADC(0)	DAC(0) ATPC_TX_REG ATPC_TX_EN	RX_PWR
temperature monitoring	100	INT_TEMP		TEMP
demodulator alarm	100	MODEM_CPM_BER		DEM
ber alarm	1	MODEM_CPM_BER	microinter_CNT	
atpc algorithm	10	ATPC_RX_READY ATPC_RX_REG	txpwr_dac_val	ATPC_LOOP RX_REM_PWR

During the execution, the two following non-functional constraints apply a) polling time precision of each component <= 10%, b) worst case response time (WCRT) per component <= 50% of polling time.

HPC specification

HLRS develops a platform for dynamic simulation of complex physical processes in industrial technological objects. The dynamic PHANTOM platform will take as input the real live data streams that can be obtained from the sensors, deployed in the technological objects as parts of their automated control system. Leveraging the real-time capabilities of the PHANTOM development framework along with the complex models provided by the experts of the targeted domain, the dynamic simulation platform development will result in a unique solution that is not yet available on the market.

The core of the PHANTOM simulation study is **Computation Fluid Dynamics (CFD)** by the open-source CFD simulation package ***Palabos*** [16]. Palabos implements Lattice-

Boltzmann CFD method to simulate the propagation of the fluid through the geometry of any object in a closed domain, which simulates the air dynamic channel. On the HLRS project partner's infrastructure, Palabos is used by its industrial customers (such as ***Porsche***) to analyse the geometry of the newly-designed cars.

Figure A- 3 illustrates all components of HPC application and the dependencies.

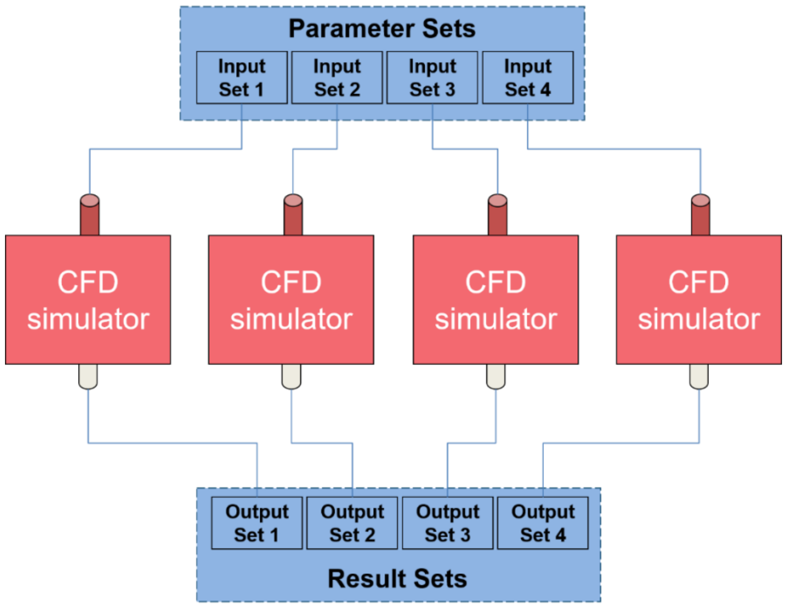


Figure A- 3 HPC application components and dependencies

The application consists of a set of PHANTOM programs, each implementing the following simulation pipeline:

1. Loading the input parameter set, consisting of:

- Geometry parameters
- Numerical parameters
    - Kinematic viscosity
    - Inlet velocity
    - Grid resolution
    - …
- Output parameters
    - Maximum simulation time
    - Frequency of the statistic output

- Type of output files (2-D, 3-D, …)

2. Performing the simulation

3. Generation of the output files

Appendix 5. REQUIREMENTS OF MONITORING LIBRARY

In the following, we describe the initial requirements (indexed as Uxx), and the new requirements detected from the questionnaires filled by the use case partners (indexed as Nxx).

Heterogeneous target platform (U73)

The PHANTOM Monitoring Framework should support all mandatory target platforms of the users' interest. To be specific, the PHANTOM infrastructure should be heterogeneous, including multi-core CPUs, GPUs, FPGAs, and the targeted embedded systems (e.g. Movidius). Subject to the hardware facilities and availabilities, sometimes a hosting hardware is required in order to monitor the connected accelerator. For instance, collecting the run-time metrics of a GPU is done by the Monitoring Framework deployed on the associated hosting CPU. The reconfigurable (FPGA) and hybrid (CPU+FPGA) device monitoring should happen via industry-standard FPGA Mezzanine Connectors (FMC), e.g. Xilinx Zynq platform.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U73	The run-time monitor shall be capable of acquiring monitoring data in all mandatory target platforms (e.g. CPU, FPGA, etc.) subject to available hardware capabilities	SHALL	YES

Metrics (U26, U72, U74-78, U83)

Generally, the PHANTOM run-time monitoring should support metrics covering both hardware (infrastructure-level) and software (application-level) properties. Some metrics are predefined, like the execution time, memory properties, power consumption, communication bandwidth, and I/O usage, while the others are application-specific metrics, which are user-defined and application-distinctive. Some examples of the user-defined metrics are the number of the processed frames for the surveillance use case or the number of the numerical integration steps for the HPC application. These predefined metrics are also different based on different hardware and sensors availabilities.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U26	PHANTOM shall provide an API for monitoring of user-defined metrics	SHALL	YES
U72	The PHANTOM run-time monitor shall be able to monitor non-functional properties of an application	SHALL	YES
U74	The PHANTOM framework should be capable of monitoring execution time properties	SHOULD	YES
U75	The PHANTOM framework should be capable of monitoring memory properties	SHOULD	YES
U76	The PHANTOM framework should be capable of monitoring power consumption properties	SHOULD	YES

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U77	The PHANTOM framework should be capable of monitoring communications bandwidth properties	SHOULD	YES
U78	The PHANTOM framework should be capable of monitoring I/O properties	SHOULD	YES
U83	PHANTOM should provide monitoring of application-specific performance metrics	SHOULD	YES

Accessibility (U35, U79, U80-U82)

The run-time monitoring accessibility requirements are mainly the following:

- Data obtained by the run-time Monitoring Framework shall be accessible to the users by some means based on the users' interest.
- Data obtained by the users should be structured in a standard format, which enables further integration requirements.
- Users should be able to control the metrics sampling frequency and to select which metrics are to be monitored.
- Data storage and historical metrics analysis shall also be provided in the Monitoring Framework.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
U35	The PHANTOM framework shall be capable of interfacing with local target platforms (deploying the application, monitoring the execution and the state of the target platform resources).	SHALL	YES
U80	The user shall be able to have fine-grained control over execution time monitoring by indicating application sub-components (i.e. tasks, loop, code blocks) whose execution time shall be monitored	SHALL	YES
U82	For non-periodic non-functional properties, the user should be able to select the frequency of data acquisition	SHOULD	YES

Requirements on the questionnaires filled by the Use Case Partners (D1.3)

After the initial round of development, Use Case partners were queried on the tools and further requirements generated. These are discussed here.

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
N1	HPC use case might need to submit more complex structure to the MF than tuples (e.g. triples), e.g. the	SHOULD	YES

Req. No.	Requirement	Overall Priority	Fulfilled (yes, partial, no)
	local timestamp of the events.		
N2	The integration of the MF in the Surveillance UC was not a straightforward process. In fact, the MF assumes that all the threads in an application are explicitly terminated, and relies on that fact to collect the default and the user-defined metrics. However, in the Surveillance UC, the devised architecture of the application includes some threads that are not terminated explicitly, instead, they terminate when the main program exits. As a consequence, the MF did not work properly at the first attempt to incorporate it into the UC; a workaround was devised and implemented, but it may not be the most elegant solution nor respect the original architecture of both the MF and the Surveillance UC.	SHOULD	YES
N3	Possibility to manage user-defined metrics without integrating the MF-client if no infrastructure metrics are required.	SHOULD	YES

Fulfillment of N1:

The requirement is solved by providing two alternative ways to the Library for registering user metrics. The new additional functions are more flexible than the available function in the Library when it was evaluated by the use cases in D1.3. The initial function allowed to define single-valued user metrics, which were composed of a label and a value like *"user_metric_label":20.5* or *"user_metric_label":"hello"*.

The first alternative is based on the user build their own JSON data structure and submit it with the function "*submit_metric_json(char * string);*", where the user is responsible for the correct syntax of the JSON string.

The user can define a string such as:

user_string ← "user_metric":{"number-of-loops":122, "us-er_timestamp":32758512}
and submit it as:

submit_metric_json (user_string);
The second alternative is based on providing a set of functions to build the JSON string from the user variables. These functions provide the capability of defining multiple fields, which can be also multivalued. The library provide new calls such generation of a new user metric "*(user_metric *) pointer = new_metric(label);*" and add fields and values to it with the calls "*add_int_field(pointer, "label", size, array_of_ints);*",

"*add_string_field(pointer,"label", size,array_of_strings);*" and "*add_time_field (pointer, "label", time_stamp);*"

Additionally, to the fields defined by the user, it is automatically registered the local timestamp when is created every single user-metric.

As an example of the capabilities and the flexibility of the new version, we can see the following metrics that can be generated:

`"user_metric_label": [20,30, 40]`	`"user_metric_label" : {`
	`  "label_a": [30,40],`
	`   "label_b": "hello",`
	`   "label_c": [ "one", "two"],`
	`   "my_time":"2018-06-28T08:27:10.851"`
	`}`

Example of the instrumentation to register metrics like in the example appear in Appendix 6.

Fulfillment of N2:

The structure of the GMV use case caused issues with the initial version of the framework. An approach to solve this was developed and is explained in Appendix 7.

The initial design of the MF Library collects the metrics locally and submit them at the end of the execution of the application. However, the requirement N2 requests to periodically submit buffered metrics because their running threads keep running for a long time. Therefore, the submission of buffered monitored metrics is supported, which can be automated with a fixed frequency in the variable ***"transfer_interval"***[2]. This additional parameter will be defined with the other initial metric parameters. As an example:

```
int main() {
 /* MONITORING METRICS */
   metrics m_resources;
   m_resources.num_metrics = 3;
   m_resources.local_data_storage = 1; //remove the file if user
      unset keep_local_data_flag

   m_resources.sampling_interval[0] = 10; // 1000 stands for 1s
   m_resources.transfer_interval[0] = 600; // 1min
   strcpy(m_resources.metrics_names[0], "resources_usage");

   m_resources.sampling_interval[1] = 10; // 1s
```

[2] Assigning a value of zero the *transfer_interval* parameter produces that the metrics be sent as described in the previous deliverables, that is, when finished the execution of the application. The impact of doing or not periodic transfers of data will be different depending on the application. Notice that value of this parameter defines the required buffer size, and the available space for the buffer is limited by the available storing space. Therefore, the *transfer_interval* value must smaller than the value that requires filling such available space for buffering.

```
   m_resources.transfer_interval[1] = 600; // 1min
   strcpy(m_resources.metrics_names[1], "disk_io");

   m_resources.sampling_interval[2] = 10; // 1s
   m_resources.transfer_interval[2] = 600; // 1min
   strcpy(m_resources.metrics_names[2], "power");

/* MONITORING START */
   const char server[]="localhost:3033";
   const char regplatformid[]="node01";
   mf_start(server, regplatformid, &m_resources);

   ....

   mf_end();
/* MONITORING SEND */
   const char appid[]="demo11114";
   const char execfile[]="hello_world";
   mf_send(server, appid, execfile, regplatformid);
}
```

User-defined metrics are forward to the server when the user calls the function `mf_send`, typically when the application finishes. However, to solve the requirement D1.3-L2 an additional function is defined which allows sending each thread buffer to the server:

`mf_usermetrics_send(server, appid, execfile, regplatformid, thread_id)`

The function requires the *thread_id* in order to identify the buffer to be sent and not interfere with the buffers being used by the other threads.

Fulfillment of N3:

Stopping the MF-Client and providing a configuration which does not request the use of any plugin will allow collecting only user-defined metrics, and this satisfies the requirements of the particular evaluation scenario for the resource-constrained devices of the Intecs use case.

Appendix 6. Examples of Registering User Defined metrics

The experience with the integration of the tools and the use case partners motivated the development of two alternative ways of generating the user metrics in JSON format. Next, both sets are illustrated with some examples.

First Set of Functions:

This set of functions allow the users the freedom to provide the metrics in JSON format according to their needs, which requires that users construct the JSON metrics without syntactic errors. One user defined JSON metric is presented below.

```
char user_string[256];//enough big to store the JSON
char *my_t = mycurrenttime_str();

sprintf(string_a,"\"user_metric\":{\"  loops\":12,\"user_timestamp\":
%s}", my_t);
submit_metric_json(user_string);

sprintf(string_a,"\"user_metric\":{\"co2\":8.24,\"user_timestamp\":%
s}", my_t);
submit_metric_json(user_string);
```

JSON strings submitted:

```
"user_metric":{"loops":12,"user_timestamp":327586205}
"user_metric":{"co2":8.24,"user_timestamp":327586205}
```

Notice that the function mycurrenttime_str provides the timestamp the absolute elapsed wall-clock time since some arbitrary fixed point in the past. It isn't affected by changes in the system time-of-day clock. This is best option when we want to register the elapsed time between two events observed on the one machine without an intervening reboot.

That timestamp is not a REALTIME timestamp because in such case it will be the machine's best-guess as to the current wall-clock, time-of-day time. Which can jump forwards and backwards as the system time-of-day clock is changed, including by NTP. Which is completely undesirable.

Second Set of Functions:

This second set is easier to use for those users that are not experts on the JSON syntax, so it is less prone to generate metrics with errors.

Below appear different examples of the instrumentation of the user-defined metrics, from single valued metric, Multivalued metric, Multiple fields metric and Multiple fields metric with a user-defined timestamp

Output	**Code**

`“user_metric”: 20`	`metric_query *) *my_metric;`
	`mymetric=` **`new_metric`**`(“user_metric”,0);`
	`int array_int[]={20};`
	`my_metric=` **`add_int_field`**`(&my_metric, “”, 1, array_int);`
	`submit_metric``(my_metric);`

Output	Code
`“user_metric”: [20, 30, 40]`	`metric_query *) *my_metric;` `mymetric=` **`new_metric`**`(“`**`user_metric`**`”,0);` `int array_int[]={20,30,40};` `my_metric=` **`add_int_field`**`(&my_metric, “”, 3, array_int);` **`submit_metric`**`(my_metric);`

Output	Code
`“user_metric” : { “alfa”: [30,40], “beta”: “hello” }`	`(metric_query *) *my_metric;` `my_metric=` **`new_metric`**`(“`**`user_metric`**`”,0);` `int array_b[]={30,40};` `my_metric =`**`add_int_field`**`(&my_metric, “beta”, 2, array_b);` `char array_c[1][10];` `strcpy(array_c[0], “hello”);` `my_metric =` **`add_string_field`**`(&my_metric, “beta”, 1, array_c);` `submit_metric(my_metric);`

Output	Code
`“user_metric” : { “alfa”: 20, “beta”: [ 30, 40 ], “time_stamp”: “2018-06-28T08:27:10.851” }`	`(metric_query *) my_metric=`**`new_metric`**`(“`**`user_metric`**`”,0);` `int array_a[]={20};` `my_metric =`**`add_int_field`**`(new_metric, “alfa”, 1, array_a);` `int array_b[]={30,40};` `my_metric =`**`add_int_field`**`(new_metric, “beta”, 2, array_b);` `struct timespec mytime;` `clock_gettime(CLOCK_REALTIME, & mytime);` `double timestamp_ms = mytime.tv_sec * 1000.0 + mytime.tv_nsec / 1.0e6;` `my_metric =` **`add_time_field`**`(new_metric, “`**`time_stamp`**`”, timestamp_ms);` **`submit_metric`**`(new_metric);`

Appendix 7. EXAMPLE OF MONITORING A MULTITHREAD APPLICATION

The purpose of this example is to facilitate the integration of the applications with the monitoring environment to those new users without experience. Therefore, this example aims to reduce the time cost of the user for said task.

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <math.h>
#include "mf_api.h"
pthread_t tid[2];
#define PI 3.14159265

const char server[]="localhost:3033";
char currentid[100]; // common UNIQUE id for register in the MF all
the threads-metrics in this execution.

void start_monitoring( const char *server, const char
*regplatformid){
/* MONITORING METRICS */
      metrics m_resources;
      m_resources.num_metrics = 3;
      m_resources.local_data_storage = 1; /*remove the file if user
unset keep_local_data_flag */
      m_resources.sampling_interval[0] = 10; // 1000 stands for 1s,
unit in ms
      strcpy(m_resources.metrics_names[0], "resources_usage");
      m_resources.sampling_interval[1] = 10; // unit in ms
      strcpy(m_resources.metrics_names[1], "disk_io");
      m_resources.sampling_interval[2] = 10; // unit in ms
      strcpy(m_resources.metrics_names[2], "power");
/* MONITORING START */
      mf_start(server, regplatformid, &m_resources);
}

void* doSomeThing( void *args){
    unsigned int i, j, angle=0, n =720;
    struct Thread_report *my_thread_report;
    pthread_t id = pthread_self();
    char component_name[100], temp_i[100], mytime[100];
    my_thread_report = (struct Thread_report *)args;
    my_thread_report->start_time=mycurrenttime();
    my_thread_report->total_metrics= 3;
    my_thread_report->user_label  = (char **) malloc(n *
sizeof(char*));
    my_thread_report->user_value  = (char **) malloc(n *
sizeof(char*));
    my_thread_report->metric_time = (char **) malloc(n *
sizeof(char*));
    for (i=0;i<my_thread_report->total_metrics;i++){
      my_thread_report->user_label[i]  = (char *) malloc(40 *
sizeof(char));
      my_thread_report->user_value[i]  = (char *) malloc(40 *
sizeof(char));
```

```
      my_thread_report->metric_time[i] = (char *) malloc(40 * 
sizeof(char));
    }
    for (i=0;i<n;i++){
      llint_to_string_alloc(mycurrenttime(),mytime);// <<-- TIME
      strcat(mytime, ".0");
      angle=(angle+1) % 360;
      for (j=0;j<my_thread_report->total_metrics;j++)
         strcpy(my_thread_report->metric_time[j], mytime); // <<-- 
TIME
      itoa(i,temp_i);
      if(pthread_equal(id,tid[0])) {
         strcpy(component_name, "first thread ... ");
         strcpy(my_thread_report->user_label[0], "n_ships_found"); 
// <<-- LABEL
         itoa((int) 10*i+5, my_thread_report->user_value[0]); 
// <<-- VALUE
         strcpy(my_thread_report->user_label[2], "sim-
ple_function_comp_a"); // <<-- LABEL
         ftoa((float) (40.0+20.0*cosf(((float) angle)* PI / 180.0)), 
my_thread_report->user_value[2],3);
      } else {
         strcpy(component_name, "second thread ... ");
         strcpy(my_thread_report->user_label[0], "num-
ber_of_blocks"); // <<-- LABEL
         itoa((int) 20*i+8, my_thread_report->user_value[0]); 
// <<-- VALUE
         strcpy(my_thread_report->user_label[2], "sim-
ple_function_comp_b"); // <<-- LABEL
         ftoa((float) (40.0+20.0*sinf(((float) angle)* PI / 180.0)), 
my_thread_report->user_value[2],3);
      }
      strcat(component_name, temp_i);
      printf("\n Processing thread named as: %s\n",component_name);

      strcpy(my_thread_report->user_label[1], "counter"); // <<-- 
LABEL
      itoa((int) i, my_thread_report->user_value[1]);     // <<-- 
VALUE
      user_metrics_buffer(currentid,*my_thread_report);
      usleep(10000);  /* sleep unit is on us */
    }
    printf("\n Finishing thread named as: %s\n",component_name);
    fflush(stdout);
    my_thread_report->end_time=mycurrenttime();
    return NULL;
}

int main(int argc, char* argv[]) {
      const int amount_of_threads = 2; //we have 2 threads in this 
example
      /*******************MONITORNG START ************************/
      struct Thread_report all_thread_reports[2];
      char regplatformid[]="node01";
      char appid[]="demo";
      char execfile[]="pthread-example";
      start_monitoring(server, regplatformid);
      strcpy(all_thread_reports[0].taskid,"component_a");
```

```
	strcpy(all_thread_reports[1].taskid,"component_b");
	for(int i=0;i<amount_of_threads;i++)
		all_thread_reports[i].total_metrics=0;
	if(argc>1)
		strcpy(currentid,argv[1]);
	else
		strcpy(currentid,"missingid");
	/************************************************************/
	for(int i=0;i<amount_of_threads;i++){
	int err = pthread_create(&(tid[i]), NULL, &doSomeThing, (void
*) (&all_thread_reports[i]));
	if (err != 0)
		 printf("\n Can't create thread :[%s]", strerror(err));
	else
		 printf("\n Thread created successfully\n");
	}
	for(int i=0;i<amount_of_threads;i++)
	  (void) pthread_join(tid[i], NULL);
	printf("\n Finishing program\n");
	/************* MONITORING END *******************************/
	register_workflow( server, regplatformid, appid, execfile);
	for(int i=0;i<amount_of_threads;i++)
		register_end_component(currentid,
all_thread_reports[i]);
	monitoring_send( server, appid, execfile, regplatformid);
	mf_end();
	for(int i=0;i<amount_of_threads;i++){
	   printf(" Execution label of the workflow: \"%s\"\n", cur-
rentid);
	   printf("   THREAD num %i, name : %s\n",i,
all_thread_reports[i].taskid);
	   printf("     total metrics
%i:\n",all_thread_reports[i].total_metrics);
	   for(int j=0;j<all_thread_reports[i].total_metrics;j++)
		printf("%s:%s\n", all_thread_reports[i].user_label[0],
all_thread_reports[i].user_value[0]);
	   printf("   Start time: %llu ns\n",
all_thread_reports[i].start_time);
	   printf("   End time   : %llu ns\n\n",
all_thread_reports[i].end_time);
	}
	/************************************************************/
	for (int i=0;i<amount_of_threads;i++){
	   for (int j=0;j<all_thread_reports[i].total_metrics;j++ ){
		free(all_thread_reports[i].user_label[j]);
		free(all_thread_reports[i].user_value[j]);
		free(all_thread_reports[i].metric_time[j]);
	   }
	   free(all_thread_reports[i].user_label);
	   free(all_thread_reports[i].user_value);
	   free(all_thread_reports[i].metric_time);
	}
	return 0;
}
```

Source code available at:

https://github.com/PHANTOM-Platform/Monitoring/tree/master/example-monitoring-app-with-pthreads