



PHANTOM

Cross-Layer and Multi-Objective Programming Approach for
Next Generation Heterogeneous Parallel Computing Systems

Project Number 688146

D3.1 – First report on programmer- and productivity-oriented software tools

**Version 2.0
2 November 2017
Final**

Public Distribution

**Unparallel Innovation, Wings ICT Solutions, Easy Global
Market, University of Stuttgart**

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

Every effort has been made to ensure that all statements and information contained herein are accurate, however the PHANTOM Project Partners accept no liability for any error or omission in the same.

© 2017 Copyright in this document remains vested in the PHANTOM Project Partners.

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 Phone: +39 050 9657 513 E-mail: silvia.mazzini@intecs.it	The Open Group Scott Hansen Rond Point Schuman 6 5 th 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	12/04/17
0.2	Contribution of WINGS to Parallelization Toolset and Programming Interface	05/05/17
0.3	Contribution of EGM to Model Based Testing section	09/05/17
0.4	Update of WINGS' and EGM's contributions	25/05/17
0.5	Contribution of HLRS to Phantom Application Parallelization Approach and of YORK to FPGA section on the parallelization toolset	30/5/17
0.6	Executive Summary, Introduction and Conclusion	31/5/17
0.7	HLRS and York review and corresponding modifications	7/6/17
1.0	Final Contributions and Modifications	9/6/17
2.0	Contributions from WINGS, EGM, YORK and UNPARALLEL of the Innovations Beyond the State-of-the-Art subsections to incorporate EC reviewers' comments.	2/11/17

TABLE OF CONTENTS

1. Introduction	1
1.1 Scope.....	1
1.2 PHANTOM Application Parallelization Approach.....	2
2. Parallelization Toolset.....	3
2.1 Use Case requirements.....	4
2.2 Code Analysis	4
2.2.1 Design Specifications	4
2.2.2 Implementation Details	6
2.2.3 Demonstration/Example usage	8
2.2.4 Dependencies/integration aspects.....	9
2.2.5 Innovations beyond the state-of-the-art.....	10
2.3 Technique Selection.....	11
2.3.1 Design Specifications	11
2.3.2 Implementation Details	12
2.3.3 Dependencies/integration	15
2.3.4 Demonstration/Example usage	16
2.3.5 Innovations beyond the state-of-the-art.....	17
2.4 FPGAs	18
2.4.1 PHANTOM Hardware Interface	20
2.4.2 Innovations beyond the state-of-the-art.....	20
3. Programming Interfaces.....	23
3.1 Use Case requirements.....	23
3.2 Shared Memory API	24
3.2.1 Design Specifications	24
3.2.2 Implementation Details	24
3.3 Queue API.....	26
3.3.1 Design Specifications	26
3.3.2 Implementation Details	26
3.4 Signal API.....	29
3.4.1 Design Specifications	29
3.4.2 Implementation Details	29
3.5 PHANTOM API for CPU-GPU communication	32
3.5.1 Design Specifications	32
3.5.2 Implementation Details	33
3.6 Dependencies/integration.....	35
3.7 demonstrator/Example usage	35
3.8 Innovations beyond the state-of-the-art.....	37
3.8.1 Background technologies utilised in development.....	37
3.8.2 Summary of new technologies/extensions developed.....	38
3.8.3 Early/Full Prototypes functionality	38
4. Model Based Testing	39
4.1 Use Case requirements.....	39

4.1.1 Surveillance specification.....	40
4.1.2 Telecom specification.....	42
4.1.3 HPC specification.....	43
4.2 <i>Design Specifications</i>	44
4.3 <i>Implementation Details</i>	45
4.3.1 MBT models.....	47
4.3.2 Generated Test cases	50
4.3.3 TTCN-3 Publisher	51
4.3.4 Codec/Decoder	53
4.3.5 System Adapter	53
4.3.1 Implementation Summary	55
4.4 <i>Demonstration and Testing Results</i>	55
4.5 <i>Dependencies/integration</i>	58
4.5.1 Integration objectives	58
4.5.2 PHANTOM platform interfaces	58
4.5.3 MBT interaction flow with PHANTOM	59
4.6 <i>Innovations beyond the state-of-the-art (EGM)</i>	60
4.6.1 Background technologies utilised in development.....	60
4.6.2 Summary of new technologies/extensions developed.....	61
4.6.3 Early/Full Prototypes functionality	62
5. Conclusion	64
6. References	65

EXECUTIVE SUMMARY

This document describes the initial developments on the tools and technologies to support the activities of the Parallelization Toolset and Model Based Testing modules of the PHANTOM architecture, and on the specification of the PHANTOM Programming Interface. Further developments will be reported in D3.2 – “Final report on programmer- and productivity-oriented software tools”.

In section 2, the Parallelization Toolset is described. Presented in this section are the technologies and algorithms for both code analysis and technique selection. Code analysis uses tools like ANTLR and CETUS to parse and to identify parallelisable code. This section also describes the methodology used to select the proper technology for the implementation of parallelised tasks based on the deployment plan provided by the Multi-Objective Mapper. The PHANTOM API is implemented by either CUDA, OpenMP, OpenCL, MPI or Pthreads APIs, based on these decisions. Still in the context of the Parallelization toolset, it is also provided some insight on the work developed for task parallelization on FPGAs.

Section 3 identifies and describes APIs to support the development of PHANTOM applications following a component-based approach. These APIs use the C programming language and allow the use of generic parallelisation functionalities, addressing both synchronization and data sharing mechanisms. This section also presents the PHANTOM API for communications between CPUs and the attached GPU devices, used to describe functions intended to be executed in GPU devices.

The final section reports the current status of Model Based Testing development. A study of the functional behaviour of each use case is performed to understand the expected inputs and outputs of the tests. A methodology for the definition and execution of tests is provided, being also identified the tools to be used in PHANTOM and how the Model Based Testing module will interact with other modules of PHANTOM architecture.

1. INTRODUCTION

1.1 SCOPE

This document reports the progress 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 the PHANTOM Programming API, the Parallelization Toolset and the Model Based Testing.

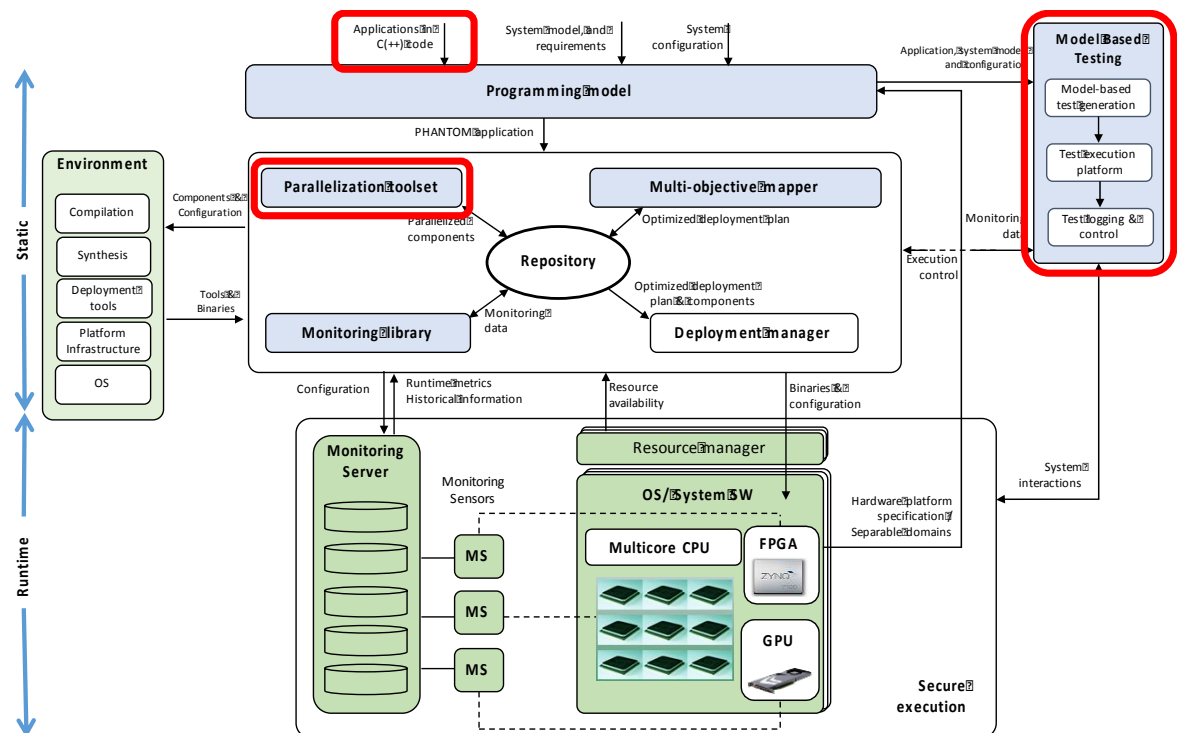


Figure 1-1: Modules of the PHANTOM architecture addressed in WP3

The Parallelization Toolset, which will be discussed in section 2, identifies parallelizable sections of the PHANTOM components, provides them to the Multi-objective Mapper and, if the mapper elects to use a parallelisation scheme for that component, implements it.

The PHANTOM Programming Interface details and implements the communication and data sharing between the different components that compose a PHANTOM application. This is described in section 3.

Model Based Testing corresponds to a toolset dedicated to performing the testing of the PHANTOM application, both helping developers to test the functional behaviour of the application and providing metrics to help the Multi-objective Mapper on the decision of parallelization plan. This component will be described in section 4.

1.2 PHANTOM APPLICATION PARALLELIZATION APPROACH

The PHANTOM programming model for applications (see D2.1 for details) follows a component-based approach – the application is constructed as a set of individual components (Figure 1-2). Components have their own thread(s) of control and are independent. They do not share data or communicate, except where explicitly enumerated by the design of the application. The enumeration of components and their shared data is called the Component Network.

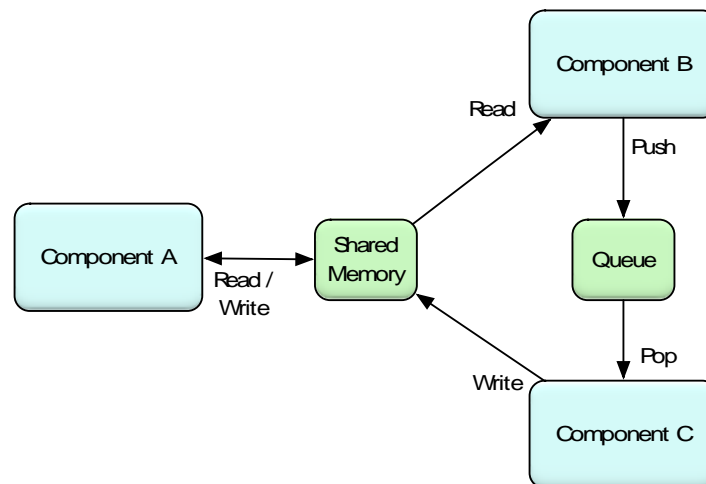


Figure 1-2: Component-based PHANTOM application

The components are separate processes and can thus benefit from deployment on distributed hardware resources (see D4.2 for details on the PHANTOM heterogeneous infrastructure testbed). The programming model provides the notion of communication channels, through which the components must communicate with one another in order to exchange the data or synchronise the execution (such as a one-directional “queue” or bidirectional “shared”, see more in D2.1).

Since the PHANTOM programs are running on different hardware and thus do not share any common compute resources, they can be treated as fully parallel executions. In order to synchronise the execution of application components in accordance with the application logic, the programs might use sync messages (which are enumerated in the Component Network).

The component-based execution constitutes a basic level of parallelism – coarse-grained. Coarse-grained parallelism depends on the application logic and is enforced by the application developer through the programming model (and the associated execution environment of the PHANTOM platform).

The next parallelisation level – fine-grained parallelism – can be achieved inside the individual components and aims to fully utilize the available parallel compute power (such as CPU cores, GPU kernels, or FPGAs). Fine-grained parallelism requires a special framework – the Parallelization Toolset.

2.

The Parallelisation Toolset is responsible for the identification of concurrent regions in the application code, for source-to-source code transformation to implement parallelisation, and for considering pre-specified requirements and the decision of the Multi-Objective Mapper described in D2.1.

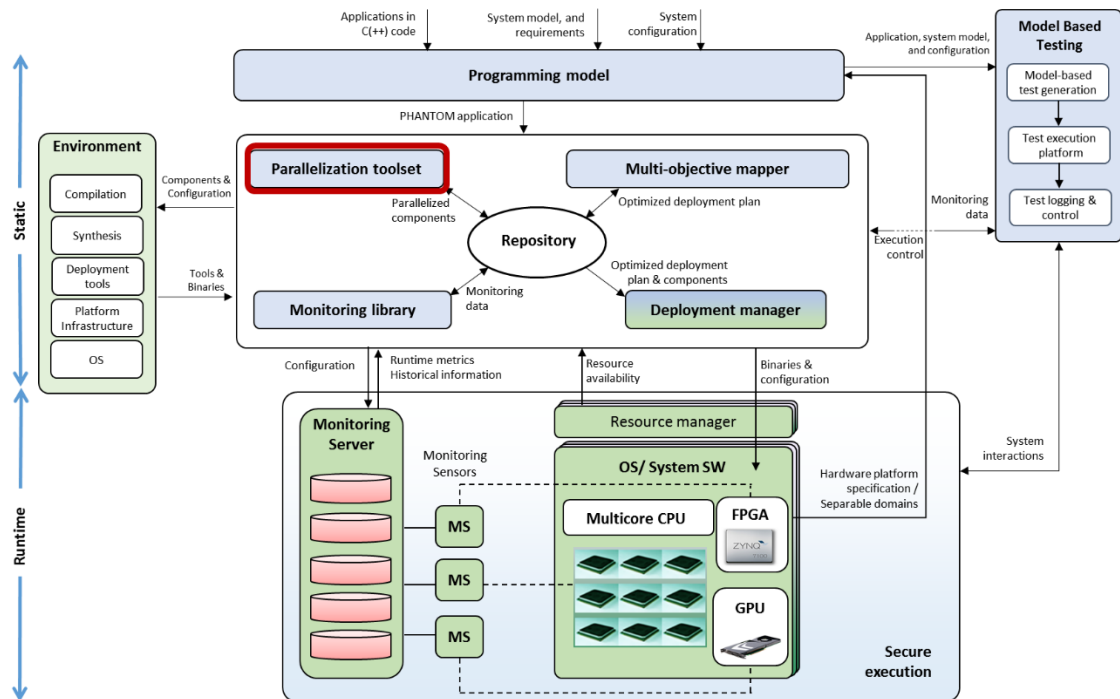


Figure 2-1: Parallelization Toolset positioning in PHANTOM toolflow.

Since PHANTOM uses a component based programming model (as described in D1.2), the Parallelisation Toolset retrieves the components which compose the PHANTOM program, analyses and transforms them in order to 1) replicate them to many identical copies that run on different ‘slices’ of input data and 2) to create different versions of components which exploit GPUs, FPGAs, SMP multiprocessors or Cloud environments. The first operation is performed by the Code Analysis described in section 2.2, which provides appropriate parallelization information to the Multi-Objective Mapper. Then the latter functionality is performed by Technique Selection (described in section 2.3), driven by the mapping decision of the Multi-Objective Mapper and assisted by four sub-toolsets, which process the components towards a specific platform architecture. The sub-toolsets consist of:

- CPU Toolset (CT) for transforming components targeting shared memory uniform memory access (UMA) or non-uniform memory access (NUMA), cache coherent, symmetric multiprocessing (SMP), CPU architectures
- GPU Toolset (GT) which transforms components for graphics processing units (GPU) implementation
- FPGA Toolset (FT), handling components for FPGA implementations
- Cloud Technologies Toolset (CTT): Transforms components for Cloud environments

This document primarily considers the CPU Toolset and GPU Toolset, whilst the rest of the sub-toolsets will be described in future deliverables of WP3 and WP4.

2.1 USE CASE REQUIREMENTS

The Parallelisation Toolset is one of the main components of PHANTOM framework and it will transform and generate parallel code for the heterogeneous platforms that comprise the infrastructure of PHANTOM. Its operation is therefore significant for all three use cases: Surveillance, Telecommunications and High-Performance Computing. In general, the use case requirements refer to the support and the capabilities of code generation, support of heterogeneous platforms, support of parallelisation APIs and programming languages, as defined in D1.1. The addressed requirements are the following:

Req. No.	Requirement	Overall Priority
U3	Parallelization of sequential application code, when complemented by parallelization instructions provided by the user	SHALL
U4	Automatic identification and parallelization of regions of sequential application code	SHOULD
U5	Support for multi-threaded concurrent tasks, including communication and synchronisation	SHALL
U6	Support of parallelization, influenced by non-functional requirements information	SHOULD
U7	Support for communications data-centric applications (e.g. automatic scaling of components to the actual size of data to be processed)	SHALL
U8	Support for component-based application design	SHALL
U14	Exploitation of SIMD instructions sets provided by CPUs	SHOULD
U19	Generation of target dependent parallel code for all mandatory target platforms without user involvement when sufficient annotations are provided.	SHOULD
U21	Automation of transferring data to/from different memories according to the component data model	SHALL
U22	Support for indication of application blocks to be parallelized	SHALL
U23	Support for indication of data dependencies, defining how data can be partitioned/split among the parallel application components	SHALL
U32	Support for application source code developed in C	SHALL
U33	Support for higher level language such as Java and C++	MAY
U37	Support for exposing the generated parallel code to the user	SHALL
U38	User modifications of the generated parallel code subject to restrictions or protected segments	SHOULD

2.2 CODE ANALYSIS

The main task of the Code Analysis is to parse the components' source code of the user provided application and perform analysis for identifying the code's parallel regions, along with transformation, with emphasis on loop and task parallelization.

2.2.1 Design Specifications

Code Analysis retrieves the Component Network (described in D1.2.) which is provided by the user and stored in the PHANTOM Repository. Code Analysis parses the Component Network in order to identify the user-provided components along with their

source code, which are stored in the Repository. Then the tool performs its processing according to the workflow depicted in the following figure:

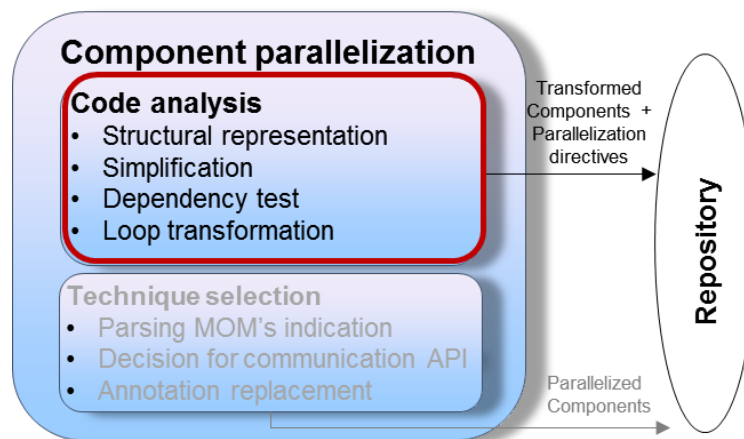


Figure 2-2: Code analysis positioning in PHANTOM tool-flow

The first operation includes the creation of a structural representation (i.e. parse tree as depicted in Figure 2-3) of the source code of each component, which will facilitate the identification of the variables and functions, important for loop and task parallelization.

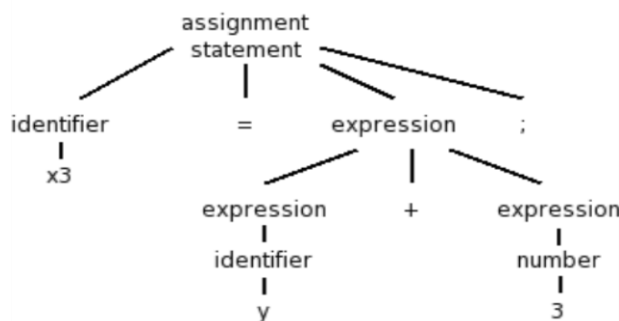


Figure 2-3: Example of source code structural (tree) representation.

Based on the structural representation, the analysis will attempt to further simplify the source code in order to further facilitate the loop and task parallelization. The code analysis is then performed by searching the simplified source code for data dependencies [10][11][12], that could prevent the components parallelization (using tools such as COINS [8], CETUS [9]). The next step is the identification of “for” loops and variables that can be parallelized, where the tool annotates them as parallelizable or not according to the dependency test outcome. In case a loop is parallelizable, the code analysis transforms the parallelizable loops-variables attributes (e.g. iteration size-limit) with specific PHANTOM directives (e.g. `phantom_slice_size()`) that enable the selected loop variables to be further parallelized and the component to be replicated.

Finally, the analysis exports the parallelization annotations/directives which provide the maximum number of possible parallel components to the Multi-Objective Mapper module (described in D2.1).

2.2.2 Implementation Details

The Code Analysis is a set of classes and functions implemented in Java, using appropriate XML libraries to be able to parse and modify XML documents. The input of the Code Analysis is the Component network XML document and the specified components' source codes (at the current stage C/C++) along with their header files that will be further analysed.

Then the Code Analysis uses ANTLR [1] (see also D4.1) tool to parse and create the structural (tree) representation of each identified component. An example of the ANTLR operation is provided in the following figure:

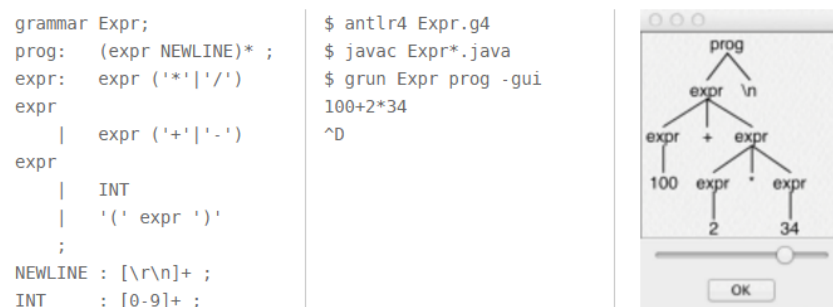


Figure 2-4: Example of ANTLR parsing. (Picture from ANTLR framework [1])

ANTLR stores the exported parse tree in a Java form that is aligned with the Java form of the Code Analysis and simplifies the following steps of the analysis.

In order to achieve a more efficient analysis of the source code elements (declarations, expressions variables), Code Analysis employs the CETUS tool [9] (see also D4.1) in order to provide an intermediate representation between the parse tree and the source code, as shown in Figure 2-5, that will consist of the objects that are more meaningful for the parallelization process (e.g. for, while, loops, loop size, loop iterators, etc.).

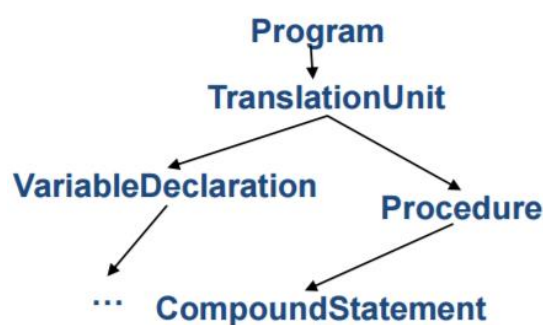


Figure 2-5: Example of intermediate representation using CETUS [9]

The next step consists of code simplification in which the Code Analysis attempts to transform the source code assignments and operations in a form that will facilitate the loop and task parallelization (e.g. simplifies mathematical operations). The Code Analysis is currently using CETUS functions for code simplification but also other refactoring tools are under investigation, such as CodeRush [3] and AutoRefactor [4]. Specifically, the Code Analysis will execute the single variable declaration function, which re-writes variable declaration to achieve single variable per declaration. Then it executes induction substitution which recognizes and substitutes induction variables in loops that take the form of $iv = iv + \text{expr}$ [9]. Assignments of this form prevent a loop from being parallelized due to its data dependence, since variables inside a loop cycle depends on the values assigned in other cycles. The CETUS induction substitution will transform these assignments to a form that does not include significant data dependencies, thus enabling loop parallelization. This substitution process is exemplified in the following figure.

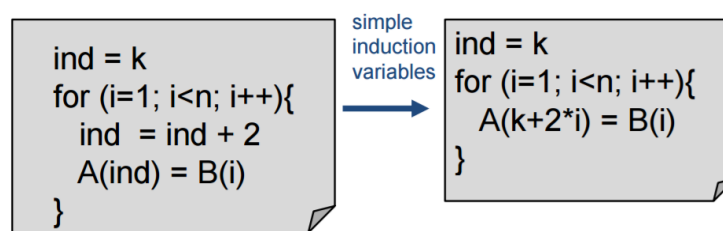


Figure 2-6: Example of code simplification

After the code simplification, the Code Analysis will perform one of the most significant tasks which is the identification of data dependencies [10][11][12]. As aforementioned, the Code Analysis will give emphasis on searching for loops that are parallelizable, since they consume most of the execution time of a sequential program. In case a loop includes dependencies between its instructions it cannot be parallelized without affecting its initial context and operation. These situations require the execution of code transformation to eliminate the data dependencies. The CETUS data dependence analysis framework [9] gathers dependence information for array accesses within loop nests and creates a data dependence graph, on top of which it performs conventional dependence tests, such as the Banerjee test [13][14] and the Greatest Common Divisor test [15]. In case that a loop includes data dependencies that cannot be resolved then the loop is characterized as not parallelizable. Some data dependencies can be resolved either by CETUS's conventional methods either by custom modifications and algorithms. The current version of Code Analysis extends the loop dependence handling by introducing a custom modification of the CETUS testing to enable loop transformation towards the PHANTOM parallelization requirements. For example, in case of a specific type of data dependence in a "for" loop that can be split by iteration size, the code analysis will slice the "for" loop in a multitude of "for" loops by substituting the iteration size with the PHANTOM specific directive (`phantom_slice_size()`), thus each component will execute the loop until the size of the each slice, as shown in Figure 2-7.

<pre>for(int i = 0; i < 1024; i++) { output_sum[i] = data[i] * (i+4) * (i-16); }</pre>	<pre>/* PHANTOM Comment: Loop# 0 is-parallel :true */ for(int i = 0; i < phantom_slice_size(); i++) { output_sum[i]= data[i] * (i+4) * (i-16); }</pre>
---	---

Figure 2-7 Example of a parallelizable loop

Data dependency test is an aspect that can improve loop parallelization and is investigated for the future versions of the Code Analysis. Since, PHANTOM addresses parallelisation of components targeting heterogeneous platforms, more data dependency algorithms will be investigated, considering also range analysis aspects for future versions.

The final step of the Code Analysis includes the exploitation of the achieved parallelization level of the analysed components. The analysis edits the Component Network XML document where, for each component, it adds the maximum number of possible parallel components (and communication objects accordingly), as described in the following figure:

```
<!-- Components Description -->
<component name="A" type="asynchronous">
    <PT:parallelisation-directive max_number="64" name="subcomponents" set-by="PT"/>
</component>
```

Figure 2-8: Example of parallelization directive

The above figure depicts an example of the XML element “parallelisation directive” which provides the maximum number of possible parallel subcomponents, for a specific component (in this case “component name=’A’ ”). This information will assist the Multi-Objective Mapper module (described in D2.1) to further replicate the parallelizable components, improving the mapping outcome and overall efficiency.

2.2.3 Demonstration/Example usage

The Code Analysis is evaluated in terms of its ability to identify parallelizable loops in the source code of the user provided components. The current testing scenario consists of the following steps:

1. The Parallelization Toolset retrieves the component network from the repository
2. The Code Analysis parses the components specified in the component network
3. The identified loops are tested for data dependencies

4. The parallelizable loops are transformed accordingly, with the addition of PHANTOM specific directives
5. The Code Analysis exports the parallelization directives to the Component Network that will be used by MOM.

The following picture provides a test run of the Code Analysis on a custom source component (in the C/C++ language), along with a Graphical User Interface that helps the user to execute the code analysis.

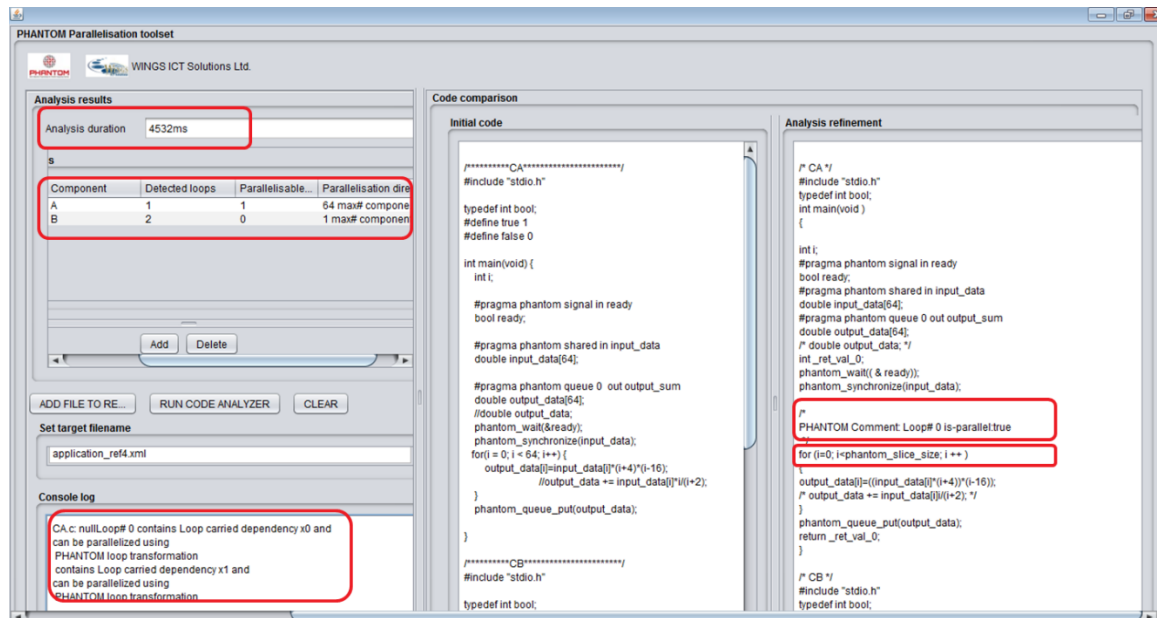


Figure 2-9: Example of Code Analysis test

In the example of the above, the analysis identified all existing loops and was able to parallelize one of them 64 times, whilst the remaining “for” loops could not be parallelized due to unresolved dependencies. Current results show that the first version of the Code Analysis is able to identify all existing loops inside the components’ source code, and is able to resolve-parallelize a number of non-complex existing loops. The number and the complexity of the parallelized loops is expected to grow in the next versions of the Code Analysis. Furthermore, the Code Analysis is tested in terms of execution time since it might add a non-trivial overhead to the overall optimization and optimization process. Current results show that the execution time of the Code Analysis has values in the range of a few seconds (<10s) for up to 5 source code components, but more tests will follow in the future implementations.

2.2.4 Dependencies/integration aspects

The Code Analysis is related to the PHANTOM Programming Model (see D1.2), from which it retrieves the user defined Component Network, along with the application components’ source code to be parallelized. Then, the Code Analysis transformation phase uses the PHANTOM Communication APIs, protocols and corresponding functions that will be added in the components’ source code to provide the loop parallelization. Finally, the Code Analysis is strongly related to the Multi-Objective

Mapper (MOM) developed in WP2, since the analysis will provide the parallelization directives facilitating MOM to explore the available number of the parallelized components, in order to replicate them. Existing parallelization tools and frameworks will be investigated to facilitate the deployment of the parallel design regions, as suggested by WP1, on the heterogeneous infrastructure, set up by WP4.

2.2.5 Innovations beyond the state-of-the-art

2.2.5.1 *Background technologies utilised in development*

XML Parsing Classes

The Code Analysis is a set of classes and functions implemented in Java, using appropriate XML libraries to parse and modify XML documents. These libraries are mainly required to process the Component Network that provides all necessary information about the components.

Source code Parser

The Code Analysis uses the ANTLR tool (<http://www.antlr.org/>) to parse and create the structural (tree) representation of each identified component. ANTLR stores the exported parse tree in a Java form that is aligned with the Java form of the Code Analysis and simplifies the following steps of the analysis.

Intermediate Representation (IR) and Dependence Analysis

CETUS compiler infrastructure (<https://engineering.purdue.edu/Cetus/>) is used for the IR and the Dependence Analysis of the components' code. The tool is responsible for both, creating an intermediate level of description for the code, and running some of the latest available dependence tests on it, determining if it's parallelizable or not.

2.2.5.2 *Summary of new technologies/extensions developed*

High-Level Annotations and Parallelization Directives

Code Analysis was extended with certain functionalities designed to provide helpful information about the components' code. In specific, high level annotations are produced by analysing the results provided by CETUS, as well as directives about components parallelization capabilities are added in the component network, information that will be used for the functionality of the Multi-Objective Mapper.

2.2.5.3 *Early/Full Prototypes functionality*

Early-First Year Prototype

The component model is being successfully extracted from the Component Network to be used by the rest of the Code Analysis' functionalities. The analysis of the components' code is able to employ successfully the CETUS compiler infrastructure and locate certain parts of the code that do not seem to have any dependencies, thus, can be parallelized. According to the information extracted by the Code Analysis, parallelization directives are successfully added to both the Component Network and to the source code of each component.

Full Prototype and Next Steps

Driven by the continuous research that is being done on automatic parallelization, other tools are being compared with CETUS, with new techniques at their disposal and more capabilities. In specific, the ROSE Compiler has already been embedded in the PT and is currently being tested against CETUS with promising results, as well as PLUTO + Polly (with its use of the Polyhedral Model). In addition, other code analysis tools will also be investigated in the context of source code simplification and to further provide more facilities in code transformation and dependence analysis, in regard to the development of the latest parallelization techniques available.

2.3 TECHNIQUE SELECTION

The Technique Selection (TS) operation is performed after the execution of the Multi-Objective Mapper, in order to receive its mapping decision and produce the parallelization indications inside the components' source code, to guide the Deployment manager on the generation or activation of the actual parallelization functions.

2.3.1 Design Specifications

The Technique Selection receives the MOM outcome, indicating the mapping decisions, along with the platform description and the component network to further decide on the best parallelization API for the parallelization technique (e.g. OpenMP, threads communication, MPI, etc.). Furthermore, TS will provide information to the PHANTOM API execution management functions, developed in the context of PHANTOM to initialize/finalize important functionalities of low level communication APIs, applied to all components that will use those APIs (e.g. in case there are more than one pthread component these functions initialize mutex variables). These functions facilitate the Deployment Manager with the adoption and execution of low level communication APIs. TS functionalities are detailed in the following figure:

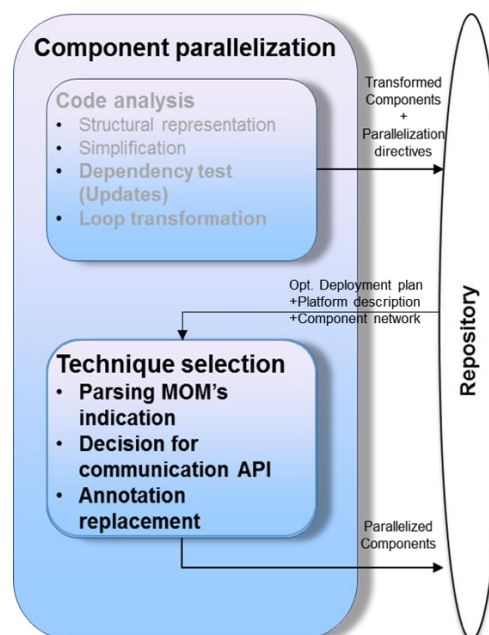


Figure 2-10: Techniques selection positioning in PHANTOM tool-flow

As depicted in the above figure (Figure 2-10) TS first parses the MOM's outcome (i.e. optimized deployment plan) in order to:

1. Classify the components depending on the type of platform to which they are mapped;
2. Identify the interacting components of the final deployment.

Then, the TS operation executes a fast and low complexity decision mechanism which selects the appropriate low-level communication API (e.g. OpenMP, MPI, OpenCL) that is the best fit for the components, according to the specified mapping to physical resources (processing elements).

Furthermore, TS provides information regarding the PT techniques selection by editing PHANTOM API functions with annotations that will facilitate the Deployment Manager operation based on existing parallelization APIs. For example, `#pragma phantom MPI_Send`

In addition, the existing communication protocol annotations (e.g. `#pragma phantom shared in data`), are replaced by lower level communication directives (e.g. `#pragma phantom MPI_Send()`), that indicate the Deployment manager to generate or activate the actual parallelization functions.

2.3.2 Implementation Details

The Technique Selection tool is implemented in Java and similarly to the Code Analysis it uses XML libraries to parse its input and generate its output. The input to TS is the Optimized Deployment Plan (in XML form) provided by the MOM, the Component Network XML document and the specified components' source codes (currently supporting C/C++) along with their header files that will be further analysed.

The first step of the operation includes the XML parsing and analysis of MOM's optimized deployment plan along with the XML parsing of the Component Network in order to correlate the information from the two documents. The current implementation considers the following component attributes:

Table 2-1: Component attributes for Technique Selection

Component Attributes
Mapping name: e.g. component_A_1_map
Mapping type: e.g. processing
Component name: e.g. A_1
Component id: 1
Subcomponents: e.g. 32

Processor name: e.g. P1
CPU-name: e.g. CPU2
Device Type: CPU-based

The first attribute is the mapping name, relative to the name of the component. Then the mapping also includes a mapping type, which indicates whether the mapping refers to a mapping of a component to a processing element (tagged as “processing”) or a mapping of a communication object (tagged as “communication”) to a physical communication buffer. In addition, TS considers the component name, id and also the “Subcomponents” attribute, which refers to the possible number of parallel subcomponents of this component. TS considers the processor name, the CPU name and the Device Type (e.g. CPU-based) to which a component is mapped.

TS specifies the communication objects and their attributes in order to assist the technique selection process (e.g. if memory type==local, use pthreads) while also assigning specific attributes to the PHANTOM protocols and APIs, used inside the components’ source code (e.g. `queue_get(var, var->source=2...)`). The considered communication object attributes are provided in the following table:

Table 2-2: Communication object attributes for the Techniques Selection

Communication object Attributes
Mapping name: e.g. communicationObjectBF1_1_map
Mapping type: e.g. communication
Component name: e.g. BF1_1
Communication object id=0
Memory name: e.g. MEM1
Memory type: e.g. local
Source name: e.g. A_1
Source id: e.g. 1
Target name: e.g. B
Target id: e.g. 0

Similar to the component attributes, the communication object attributes are the mapping name, relative to the name of the communication object, the mapping type which in this case it is “communication” the component name referring to the communication objects name and the id of the object. In addition, TS considers the physical memory’s name (or channel/buffer) to which the communication object is mapped, along with its type (local or shared). Furthermore, TS needs to know the source component name along with its “Source-id” and the target name along with the “Target-id”.

The second step includes the parsing of components' source code, in order to specify the attributes of the variables that have to be processed (e.g. sent/received) through the PHANTOM functions. This operation identifies the following attributes:

1. The variables to be pushed in queues or memories (e.g. `#pragma phantom queue out | output_image_L |`)
2. The variable's attributes: type, dimensions, size (e.g. `double | output_image_L | [64] |`)
3. Identification of communication protocols inside the components (e.g. `phantom_queue_put(...)`)

In the third step, TS decides on the specific low-level communication API for each component (e.g. components A and B will use MPI but component C will use CUDA). The selection process iterates through the communication objects, where for each of them identifies the source (and target components) and selects the appropriate low-level communication API according to the communication type (e.g. memory/Ethernet), source processor type, target processor type. The following listing provides an example of a low-level communication API indication:

```
If (source processor type == CPU) and (target processor type == CPU) and (communication
type == local memory)
{
    return phantom_Pthreads;
}
```

Listing 2-1: Example of low level communication API

The current version of TS considers the pthreads, OpenMP, MPI, CUDA, OpenCL APIs (also described in D4.1) for the low-level communication APIs.

The final step includes the attribute replacement and code generation inside the components' source code but also in PHANTOM API headers and functions. In this direction, TS parses the PHANTOM API header files and replaces component-specific attributes with all the information derived from XML and components' code parsing, in order to provide the PHANTOM API communication functions and protocols, with appropriate information, regarding their low-level implementation. The following listing provides an example where pthreads, OMP and MPI specific attributes were added to the component.

```
//Communication objects IDs and features
#define PHANTOM_NUMOFCOMMS 2
static int phantom_source_id[PHANTOM_NUMOFCOMMS]={ 1,2};
//The direction of the communication object: 0.IN-pull, 1.OUT-push, 2.INOUT-both/update
static int phantom_direction[PHANTOM_NUMOFCOMMS]={ 0,1};

//number of components that use a relevant toolkit
#define PHANTOM_PTHREADS_COMPS 3
#define PHANTOM_OMP_COMPS 0
#define PHANTOM_MPI_COMPS 0
```

Listing 2-2: Sample of component attributes definitions

The above information is stored in specific structures initialized at the execution of the PHANTOM application and is used for the proper parallelization of the components and the proper execution of the PHANTOM API functions. The following listing shows an example of a structure to store this information.

```
struct phantom_componentlist
{
    int id;
    int dev_prc_type;
    int ext_api;
    int cmpprocess; //The process/processor in which the component belongs
    int cmp_slice_size;
    int cmp_offset;
};
```

Listing 2-3: Example of PHANTOM API structure

Finally, the existing communication protocol annotations (e.g. `phantom_queue_put (data...)`), are replaced by low-level communication directives (e.g. `cudaMemcpy (data...)`), that indicate the Deployment Manager to generate or activate the actual parallelization functions. TS replaces the arguments of the PHANTOM API functions with the specific communication object attributes, in order to match the function with the appropriate structure corresponding to the variable (e.g. `output_image_L`) and derive its attributes (e.g. `source`, `target`, `type...`). For GPU code implementation, specific low-level commands are considered in order to be generated inside the components' code, used to forward the variables and the functions to the GPU device (e.g. `cudaMemcpyAsync (dev_in, simage, size_in*sizeof(double), cudaMemcpyHostToDevice, stream[cmpid]);`).

2.3.3 Dependencies/integration

TS first interacts with the PHANTOM Repository from where it retrieves its input documents and source code. The first part of the input, consists of the Component Network and the components' source code. TS also interacts with the Multi-Objective Mapper in order to receive its outcome, defined as the optimized deployment plan. TS uses the PHANTOM APIs to select, generate or replace the high-level communication

APIs with appropriate low-level communication APIs and indications. Finally, TS sends the parallelized components to the Deployment Manager for further annotation replacement, code and metadata generation.

2.3.4 Demonstration/Example usage

TS decides on the best low-level communication API per component, but it also matches the final deployment plan with the components and their communication functions. In addition, TS replaces existing annotations and information with low-level annotations referring to low-level communication APIs, facilitating the final deployment of the parallelized components. In this direction, the current version is evaluated in terms of its ability to select the appropriate low-level communication API, according to the MOM outcome but also in terms of appropriate annotation replacement.

The following paragraph includes an evaluation scenario in which MOM has decided to map three components of an example application to the same processor, on nearby CPUs to avoid the communication overhead. Technique Selection is expected to:

1. Identify the interacting components to provide PHANTOM APIs with relative information (annotation replacement and generation) about the component's number and the source and target of the communication objects;
2. Select an API that will introduce minimal communication overhead such as OpenMP or pthreads; and
3. Generate appropriate low-level communication APIs.

The following figure depicts a diagram that includes the MOM outcome with its components and memories' mapping.

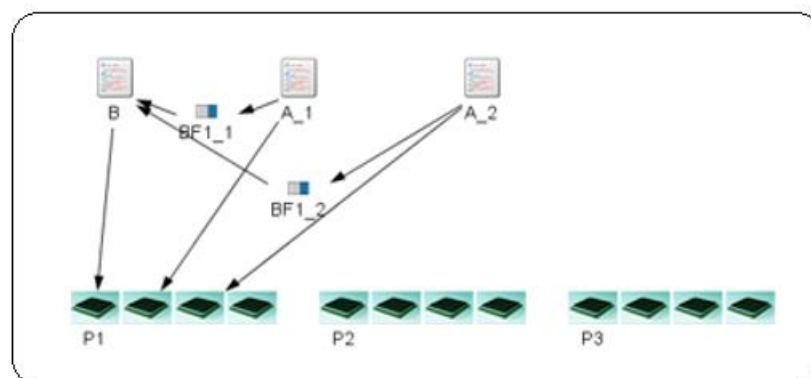


Figure 2-11: Mapping example for Technique selection demonstration

Since the components (in this case A_1, A_2, B) are mapped to the same processor and the communication objects (in this case BF1_1 and BF1_2) are mapped on two local memories, TS decides (based on the processor type and the type of the communication type) that each component will run on a different thread using pthreads. To this purpose,

TS generates the appropriate values and files in the PHANTOM API functions and communication protocols, as exemplified below:

```
#define PHANTOM_PTHREADS_COMPS 3
...
#define PHANTOM_NUMOFCOMPS 3
...
static int phantom_source_id[PHANTOM_NUMOFCOMMS]={1,2};
static int phantom_target_id[PHANTOM_NUMOFCOMMS]={0,0};
static int
phantom_memptypemat[PHANTOM_NUMOFCOMMS]={phantom_localmem,phantom_loc
almem};
```

Listing 2-4: Techniques selection attributes specification and replacement

In addition TS parses components for phantom_synchronize() functions and adds the communication object ID in order to use its attributes (source, target, low level communication API, etc.) for the low level protocols activation:

```
static int
phantom_comp_external_api[PHANTOM_NUMOFCOMPS]={phantom_Pthreads,phanto
m_Pthreads,phantom_Pthreads};

phantom_synchronize(&output_image_L,phantom_cmpid, phantom_commid);
{
    phantom_compx[phantom_cmpid]->ext_api = phantom_Pthreads;
    pthread_mutex_lock(&phantom_pth_ready_mutex);
    ...
}
```

Listing 2-5: Techniques Selection annotations replacement with low level communication API directives

2.3.5 Innovations beyond the state-of-the-art

2.3.5.1 Background technologies utilised in development

XML Parsing Classes

Like the Code Analysis, the Technique Selection is a set of classes and functions implemented in Java, using appropriate XML libraries to parse and modify XML documents. These libraries are mainly required to process the Component Network, that provides all necessary information about the components, and the mapping that is provided by the MOM.

2.3.5.2 Summary of new technologies/extensions developed

Technique Selection was fully custom made and is responsible to choose the communication technology to be used for the communication between the different code components. The main part concentrates on extracting all the information relevant to the communication objects from the component network and the mapping provided by MOM, as well as feeding this in the header files that are going to be used for the application's implementation by the Deployment Manager. Based on these header files,

the selection of a suitable communication library is performed to determine the best communication API (OpenMP, MPI, CUDA, OpenCL) that will be used in the deployment phase.

2.3.5.3 *Early/Full Prototypes functionality*

Early-First Year Prototype

Technique Selection is able to analyse the component network and the mapping of the code components, in regard to the successful choice among the various communication APIs.

Full Prototype and Next Steps

The fully developed TS will be able to insert lower level annotations, according to the chosen API, for guiding the Deployment Manager to implement (in code) the communication interfaces. A more complex and flexible algorithm will be used for determining the best-suited library for the components' communication needs.

2.4 FPGAs

The final supported technique is component parallelisation and acceleration through the use of FPGAs. The focus of the chosen approach is to ensure that modular compilation can be supported by the PHANTOM platform, and that non-FPGA experts can effectively exploit FPGAs that are present in the target architecture.

Creating FPGA designs is difficult, and requires the effort of a skilled hardware designer. It is not yet currently possible to automatically generate high-quality FPGA hardware from a software-based input, however, considerable work is being done in this area. The Xilinx SDAccel [6] development environment attempts to map GPGPU-style programs to FPGA designs, specifically transforming OpenCL programs. SDAccel does not yet target Zynq devices, only FPGA hardware designs are created without accompanying software. Zynq support is in development but not yet released. Similarly, Altera/Intel are developing support for transforming OpenCL programs to their FPGA systems. In both of these cases, the designed hardware is heavily restricted by the OpenCL programming model, and the created designs are very similar to GPGPU-style implementations. This is very effective for some algorithms, but a poor choice for others. FPGA design experts are required to operate this software, because whilst they automate some aspects of the design work, software to hardware high-level synthesis is still in its infancy, and general C/C++ code translates rather poorly. It is necessary to develop code specifically targeted for high-level synthesis rather than expecting it to translate normal software projects to FPGA designs. These tools should be therefore viewed as tools to improve the effectiveness of FPGA experts, rather than to allow non-experts to use FPGAs.

The PHANTOM project does not attempt to repeat these efforts and aims to allow their use as they become commercially-available. Instead, whilst the commercial tools catch up in these areas, and in order to promote the use of FPGA platforms by non-experts, custom IP cores will be developed manually for the Use Cases and an IP core marketplace feature is developed. These IP cores ascribe to a common set of interfaces

defined in this project, and so allow the the PHANTOM platform to automatically integrate varying sets of IP cores into new FPGA designs, thereby supporting a wide range of different target FGPA.

The PHANTOM platform supports acceleration on Xilinx Zynq FGPA [5]. The Zynq System on Chip is a multicore ARM device with a closely-coupled FPGA. This allows standard ARM software (such as Linux) to run on the CPUs whilst custom hardware executes within the reconfigurable logic. This means that for the purpose of the PHANTOM project, an FPGA target is a multicore CPU with attached accelerator, the same computation model as is used for the GPU target. An accelerated PHANTOM component consists of not just an IP core, but the ARM software that reads its data and processes the results.

In order to ensure that modular compilation of FPGA components is possible, it is necessary to ensure that any interfaces for hardware and software are defined. These are shown in Figure 2-12.

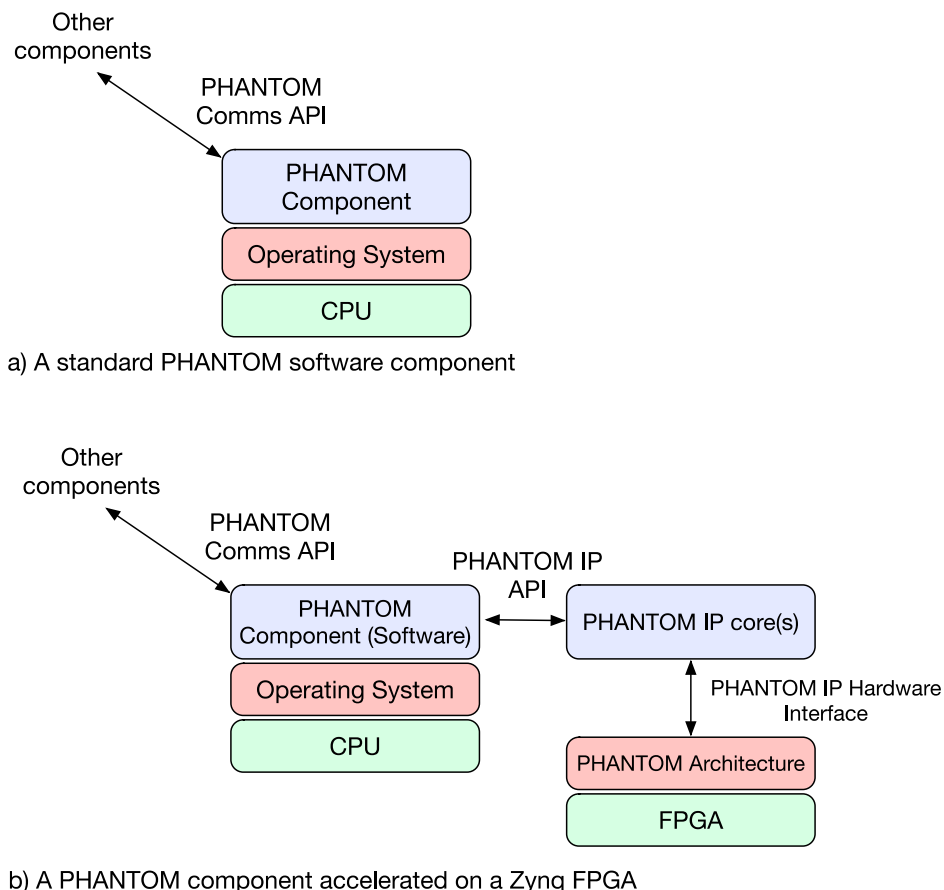


Figure 2-12: The interfaces in the PHANTOM FPGA architecture

There are two very low-level interfaces, which ideally would remain invisible to the end user. Currently they must be exposed because automatic transformation to FGPA is not yet available and must be performed manually.

The PHANTOM IP API is used by the software part of the IP which is still executing on the ARM core. This API is used to query the IP cores that are present on the FPGA, start and stop their execution, and to pass data to and from them. This API is a Linux userland API and is part of the Open Source release [7]. The current version of the API can be found at <https://github.com/PHANTOM-Platform/PHANTOM-FPGA-Linux/tree/master/API>.

The PHANTOM IP Hardware Interface defines the physical connections (wires) that a valid IP core may include to be part of a valid design. This is described in the following section.

2.4.1 PHANTOM Hardware Interface

It is often the case that multiple IP cores are to be hosted on the same physical FPGA. For example, if multiple components have hardware implementations, the Multi-Objective Mapper may choose to combine two or more IP cores. The platform therefore must be able to automatically combine IP cores into a single FPGA design. It can do this because IP cores are only allowed to have the following interfaces:

- One clock input. All IP cores run from the same system clock.
- One active-high reset line. This can be asserted by software running on the ARM cores to reset the IP core.
- One active-high interrupt line. This is used by the IP core to signal the ARM cores.
- One AXI Slave interface. This is a standard peripheral interface which allows for low-speed communications between the Linux kernel running on the ARM CPUs and the IP core.
- Between 0-4 AXI Master interfaces. This is a high-speed interface which allows the IP core to read and write main system memory. This is useful for moving data at high speed into and out of the IP core.

These interfaces are all standard, and follow the exact same format as IP cores generated from Xilinx's own software. The software which combines IP cores into a single architecture is part of the Linux software distribution. See its documentation for further details.

2.4.2 Innovations beyond the state-of-the-art

2.4.2.1 *Background technologies utilised in development*

The IP cores developed make initial use of the FPGA vendor high-level synthesis tools (Xilinx Vivado HLS) for hardware generation, but the base output is very inefficient so significant modifications must be made to create an effective implementation.

That was the case of the development of the DWT (Discrete Wavelet Transform) IP Core, which started by a generic C implementation as the input of Xilinx Vivado HLS. Despite being generated a useable IP core, the performance was sub-par, so some optimizations were needed to take fully-advantage of the usage of FPGA.

Taking the following table in consideration, it is possible to observe the magnitude of optimization possible to be achieved by having a hardware engineer performing the optimisation of the IP Core.

Table 3. Iterations of optimizations performed over the automatically generated DWT IP Core

Design	Pixels	Time (s)	Time/pixel (ns)
1	3686400	4.121374	1118
2	3686400	0.622383	169
3	3686400	0.548814	149
4	3686400	0.519238	141
5	3686400	0.469334	127
6	3686400	0.288320	78
7	3686400	0.289161	78
8	3686400	0.252271	68
9	58982400	2.991100	51
9	132710400	6.726555	51
10	132710400	5.56222	42
12	58982400	0.813501	14
13	132710400	1.188941	9
14	235929600	1.477583	6
15	235929600	1.187663	5

This shows, the poor performance of the direct generation of IP cores by Vivado HLS. And despite that human optimization not being a goal of the project, it clearly shows the level of optimization that is possible to be achieved.

2.4.2.2 *Summary of new technologies/extensions developed*

Due to the inefficiency of vendor high-level synthesis tools, it was necessary to develop interfaces to allow for modular FPGA composition of hardware components. 'PHANTOM IP Cores' were defined, which are IP cores that conform to the following developed interfaces:

- A hardware interface which specifies the I/O for the IP core, including the bus connections, and issues such as memory spaces, interrupts, and clock signals.
- A software interface which specifies how userland software in the PHANTOM Linux distribution can interact with the IP core.

By conforming to these interfaces, the FPGA infrastructure described in D4.2 can automatically create FPGA designs by integrating multiple PHANTOM IP cores, without the developer having to use the FPGA vendor tools, or even to necessarily know that the FPGAs are being targeted. This gives the platform the freedom to explore different hardware mappings of components, as long as a suitable IP core exists. Also, the use of a consistent interface ensures that once vendor high-level synthesis tools are sufficient for truly automatic use, they can be seamlessly integrated into the platform to auto-generate IP cores.

This created an additional step, and a challenge in the development of IP Cores, which is the usage of PHANTOM interfaces. Which implies going from the typical approach of FPGA/Linux integration which focus in AMBA (Advanced Microcontroller Bus Architecture), to FPGA IP cores integration with Linux via Userspace I/O.

2.4.2.3 *Early/Full Prototypes functionality*

Early-First Year Prototype

- The IP core interfaces are defined and released.
- As described in deliverable D4.2, the PHANTOM platform includes an implementation of the required hardware to software interface libraries, and of the PHANTOM communications libraries to ensure that components can communicate with the rest of the system.
- The platform also implements the automatic generation of hardware designs from a list of PHANTOM IP cores.

Full Prototype and Next Steps

- Addition of more data movement primitives to the IP core software API to more easily support efficient DMA.
- Further integration of hardware features related to security and additional monitoring.
- Experiment with integration of vendor high-level synthesis tools (Xilinx Vivado HLS) of the possibility to automatic generate (even with low performance) PHANTOM compatible IP Cores.

3. PROGRAMMING INTERFACES

The programming interfaces assist the development of component-based applications, include specific directives about parallelization, and describe functional and non-functional requirements. The provided APIs use the C programming language, enabling the incorporation of parallelization APIs (such as pthreads, OpenMPI, also described in D4.1), whilst also providing an abstraction of the system architecture, hiding the complexity between hardware and applications. Furthermore, the programming interfaces will use the PHANTOM directives defined in D1.2 to forward the user-defined functional and non-functional requirements to the implemented API functionalities. The deliverable D1.2 includes an initial description of the APIs and protocols that will be used in the context of PHANTOM, whilst more protocols and APIs may be useful to be considered in the next steps of the API's implementation.

The first version of the programming interfaces is focused on the design and implementation of communication API functions for CPU and GPU platforms in relation to code analysis and transformation operations developed in Section 2.2.5.1. Specifically, the considered APIs are the following (defined in D1.2):

- Shared API (e.g. `size_t phantom_slice_size(void *item);`) for manipulating data in the shared memory
- Queue API (e.g. `bool phantom_queue_get(void *queue);`) for manipulating blocking FIFO data items in mainly distributed memories
- Signal API (e.g. `bool phantom_notify(void *signal);`) for coordination and execution of other components, without sharing any data

The low-level APIs that are considered by the above functions are the pthreads, OpenMP, OpenMPI regarding CPU-CPU communication and CUDA regarding CPU-GPU communication (see D4.1 for details). The design and implementation of the APIs will be continuously refined and modified towards the Use Case requirements along with the progress of the development process. To this purpose, the final implementation of the APIs will be provided at the next deliverable of WP3, along with the rest of the APIs, defined in D1.2.

3.1 USE CASE REQUIREMENTS

The programming interfaces are applied in all three PHANTOM Use Cases, which will include them accordingly to the proposed Use Case applications, in order to provide the development process with appropriate functionalities, for components' parallelization and communication among the underlying heterogeneous infrastructure. In this direction, all PHANTOM Use cases will require:

Req. No.	Requirement	Overall Priority
U5	Support for multi-threaded concurrent tasks, including communication and synchronisation	SHALL
U7	Support for communications data-centric applications (e.g. automatic scaling of components to the actual size of data to be processed)	SHALL
U8	Support for component-based application design	SHALL

U19	Generation of target dependent parallel code for all mandatory target platforms without user involvement when sufficient annotations are provided.	SHALL
U20	Provision of constructs or abstractions to deal with non-uniform and uniform memory, hiding the underlying data transfer details	SHALL
U21	Automation of the process of transferring data to/from different memories according to the component data model	SHALL
U24	Provision of means for the developer to describe the composition of hardware components and interactions for the target platform	SHALL
U28	Provision of a data model for specification of input and output data	SHALL
U32	Support for application source code developed in C	SHALL
U33	Support for higher level language such as Java and C++	MAY
U47	Support for Telecom specific application classes where domain-specific libraries are commonly utilised	SHALL
U86	Support for application specific communication bus/protocols	MAY

3.2 SHARED MEMORY API

3.2.1 Design Specifications

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); (1)
```

which causes the local view of the shared memory to be updated.

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.

3.2.2 Implementation Details

The main functionality of the Shared API is focused on the “synchronize” function which is automatically implemented, according to the MOM’s optimized deployment plan and the Parallelization Toolset’s Technique Selection. The selected API implementation is then generated (or activated) by the Deployment Manager according to the selected low-level API and the processor’s type to which the component is mapped (e.g. since the component with `id=0` has `phantom_Pthreads` indication and its processor type is CPU, the Deployment Manager will generate (or activate) the function that executes `mutex_lock()`, update of shared variable and `mutex_unlock()`).

The first version of the synchronization function is provided in the following form:

```
bool phantom_synchronize(void **item, int phantom_cmpid,  
int phantom_commid) (3)
```

Which includes the following inputs:

- item: the variable to be updated;
- phantom_cmpid: is the id of the current component. Used for lower-level API generation/activation. Not modifiable by user or Phantom;
- phantom_commid: is the id of communication object. Technique selection will replace it with actual value.

The following listing provides a first representation of the actual implementation, without details, since this form may change and be refined according to the PHANTOM development process:

```
if (((ext_api == phantom_Pthreads)) && ((dev_prc_type == PHANTOM_CPUSYS)))  
{  
    pthread_mutex_lock(&phantom_pth_ready_mutex);  
  
    //Update - global variable  
    phantom_threadupdate_sync(item, phantom_cmpid, phantom_commid);  
  
    pthread_mutex_unlock(&phantom_pth_ready_mutex);  
}  
else if (((ext_api == phantom_OpenMP)) && ((dev_prc_type ==  
PHANTOM_CPUSYS)))  
{  
    #pragma omp critical  
    {  
        //Update - global variable  
        phantom_threadupdate_sync(item, phantom_cmpid, phantom_commid);  
    }  
    //printf("phantom OpenMP\n");  
}  
else if (((ext_api == phantom_MPI)) && ((dev_prc_type == PHANTOM_CPUSYS)))  
{  
    MPI_Allgather(&item, comprsize, MPItype, &memvar, comprsize, MPItype,  
MPI_COMM_WORLD);  
    //printf("phantom MPI\n");  
}
```

Listing 3-1: First representation of PHANTOM synchronize function

Furthermore, Shared API includes the function that returns the components' slice size and is currently implemented in the following form:

```
size_t phantom_slice_size(int compid); (4)
```


The input of the function is the id of the component that is sliced. It is assumed that the Parallelization Toolset's Techniques selection created specific variables that hold information such as component's communication API and component's slice size. The latter is called inside the `phantom_slice_size()` (e.g. `temp_slice_size = phantom_compx[cnum] -> cmp_slice_size`) which then returns the component's size.

3.3 QUEUE API

3.3.1 Design Specifications

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 *queue);    (5)
```

Used for pulling items from the queue:

```
bool phantom_queue_put(void *queue, void *item);    (6)
```

Used for adding items to the queue:

```
uint32_t phantom_queue_count(void *queue);    (7)
```

Used for counting the number of items currently in the queue:

3.3.2 Implementation Details

The implementation of the Queue API includes low-level communication APIs targeting mappings of components to platforms with distributed physical memories, while also considering functions able to handle thread-based APIs in case of mapping to shared physical memories. The implementation of the Queue API functions changes automatically, according to the platform that the communication is mapped. For example, when components are mapped to different devices, they will have to use a communication API supporting communication between distributed memories, such as MPI, while in case the components are mapped in the same device, components could use a low-overhead thread-based API such as pthreads (see D4.1 for further information). The actual implementation of the functions is decided by the PT's Technique Selection, using MOM's outcome and generated by the Deployment Manager. The following paragraphs describe the first implementation of the Queue API functions:

The function that is used to get data from a receiving queue is declared as follows:

```
bool phantom_queue_get(void **queue, int phantom_cmpid, int
                        phantom_commid);    (8)
```


The inputs of the function are the following:

- Queue: The variable/ array/ structure to be retrieved from the received queue
- phantom_cmpid: id of the component that calls the function.
- phantom_commid: id of communication object (in relation to the optimized deployment plan).

The first sample of the implementation is provided in the following listing, skipping details from custom source code, since this form may change and refined according to the PHANTOM development process:

```
if ((ext_api == phantom_Pthreads))&&(( dev_prc_type == PHANTOM_CPUSYS))
{
    errlock = pthread_mutex_lock(&phantom_pth_ready_mutex);
    //update queue
    phantom_threadupdate_get(queue, phantom_cmpid, phantom_commid);
    errlock = pthread_mutex_unlock(&phantom_pth_ready_mutex);
}
else if (((ext_api == phantom_OpenMP))&&(( dev_prc_type == PHANTOM_CPUSYS)))
{
    #pragma omp critical
    {
        phantom_threadupdate_get(queue, phantom_cmpid, phantom_commid);
    }
}
else if (((ext_api == phantom_MPI))&&(( dev_prc_type == PHANTOM_CPUSYS)))
{
    MPI_Status phantom_mpi_status;
    MPI_Recv(&queue, comprsize, MPItype, src_id, phantom_commid,
    MPI_COMM_WORLD, &phantom_mpi_status);
}
```

Listing 3-2: First representation of PHANTOM queue get functionality

As depicted in the above listing, in case the component with id=1 has phantom_MPI indication and its processor type is CPU, the Deployment Manager will activate or replace the phantom_queue_get with the corresponding MPI_Recv(...) function to receive the specified variable.

Similarly, the function to put data, to be sent over a queue is declared as follows:

```
bool phantom_queue_put(void **queue, int phantom_cmpid, int
                        phantom_commid); (9)
```

The phantom_queue_put() is implemented in a similar manner, where instead of thread_get and MPI_Recv functionalities, the queue_put is using thread_put and MPI_Send functions.

In addition, the function that provides the number of the items in a queue is declared as:

```
uint32_t phantom_queue_count(void **queue, int phantom_cmpid, int phantom_commid)
```

The inputs of the function are the following:

- Queue: The variable/ array/ structure to be checked for its size/number
- phantom_cmpid: id of the component that calls this function.
- phantom_commid: id of communication object (in relation to the optimized deployment plan).

The following picture depicts a first representation of the function's implementation:

```
if (((ext_api == phantom_Pthreads)) && ((dev_prc_type == PHANTOM_CPUSYS)))
{
    return phantom_chekccount(phantom_commobjx[phantom_commid]);
    //printf("phantom_Pthreads \n");
}
else if (((ext_api == phantom_OpenMP)) && ((dev_prc_type == PHANTOM_CPUSYS)))
{
    return phantom_chekccount(phantom_commobjx[phantom_commid]);
    //printf("phantom OpenMP\n");
}
else if (((ext_api == phantom_MPI)) && ((phdev_prc_type == PHANTOM_CPUSYS)))
{
    MPI_Status phantom_mpi_status;
    MPI_Get_elements(&phantom_mpi_status, phantom_commobjx[phantom_commid]-
>MPItype, &phantom_count);
    //printf("phantom MPI\n");
}

uint32_t phantom_chekccount(void** item)
{
    int i=0;
    while(item[i] != NULL)
    {
        i++;
    }
    return i;
}
```

Listing 3-3: First representation of PHANTOM queue count functionality

3.4 SIGNAL API

3.4.1 Design Specifications

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);    (10)
```

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

```
bool phantom_notify(void *signal);  (11)
```

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

```
bool phantom_notifyall(void *signal); (12)
```

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); (13)
```

3.4.2 Implementation Details

Since Signal API complements Shared and Queue API, it is implemented in a similar automatic approach, based on the MOM's optimized deployment and the Technique Selection decision which indicates the Deployment Manager for a specific low-level signalling implementation. Specifically, the function that waits for a specific signal to continue/unblock its operation has the following form:

```
bool phantom_wait(void *signal, int phantom_cmpid); (14)
```

The function consists of the following inputs:

- signal: The signal which will unblock the thread/process
- phantom_cmpid: the id of the component that calls the function

The function implementation is depicted in the following listing:

```

if (((phantom_compx[phantom_cmpid]->ext_api == phantom_Pthreads){

    pthread_mutex_lock(&phantom_pth_ready_mutex);

    while ( phantom_pth_ready_signal==0) {
        pthread_cond_wait(&phantom_pth_ready_cond, &phantom_pth_ready_mutex);
    }

    pthread_mutex_unlock(&phantom_pth_ready_mutex);

}else if (((phantom_compx[phantom_cmpid]->ext_api == phantom_OpenMP){

#pragma omp critical
{
    #pragma omp flush(phantom_omp_thread_ready)

    while(phantom_omp_thread_ready==0)
    {
        #pragma omp flush(phantom_omp_thread_ready)
    }

} else if (((phantom_compx[phantom_cmpid]->ext_api == phantom_MPI)){

    MPI_Status phantom_mpi_status;
    MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &phantom_mpi_status);
} ...

```

Listing 3-4: First representation of PHANTOM wait functionality

It must be noted that the details are skipped for the sake of simplicity and because the actual implementation will be modified according to the progress of the development process.

Similarly, the function that unblocks a thread or process or a thread has the following form:

```
bool phantom_notify(void *signal,int phantom_cmpid);    (15)
```

with the following inputs:

- signal: The signal which will unblock the thread/process
- phantom_cmpid: the id of the component that calls the function

The function implementation is depicted in the following listing:

```
if (((phantom_compx[phantom_cmpid]->ext_api == phantom_Pthreads))&&((phantom_compx[phantom_cmpid]->dev_prc_type == PHANTOM_CPUSYS)))
{
    pthread_mutex_lock(&phantom_pth_ready_mutex);

    pthread_cond_signal(&signal);

    pthread_mutex_unlock(&phantom_pth_ready_mutex);
    //printf("phantom_Pthreads \n");
}
else if (((phantom_compx[phantom_cmpid]->ext_api == phantom_OpenMP))&&((phantom_compx[phantom_cmpid]->dev_prc_type == PHANTOM_CPUSYS))
{
    phantom_omp_thread_ready=1;
    #pragma omp flush(phantom_omp_thread_ready)
    printf("phantom OpenMP #pragma omp critical \n");
}
else if (((phantom_compx[phantom_cmpid]->ext_api == phantom_MPI))&&((phantom_compx[phantom_cmpid]->dev_prc_type == PHANTOM_CPUSYS)))
{
    MPI_Send(&signal, 1, MPI_UNSIGNED_SHORT, phantom_commobjx[i]->trgt_id, i, MPI_COMM_WORLD);
}
```

Listing 3-5: First representation of phantom_notify function

The phantom_notifyall is implemented in a similar manner, with minor modifications in order to affect all the running threads or processes.

Furthermore, PHANTOM provides the barrier function

```
bool phantom_barrier(int component_id); (16)
```

which receives as input the id of the component which calls the function. This function will hold the execution of a program up to the point that is called and wait until all threads or processes, before that call have finished their work. The implementation of the function is provided in the following listing:

```

if (((phantom_compx[phantom_cmpid]->ext_api == phan-
tom_Pthreads))&&((phantom_compx[phantom_cmpid]->dev_prc_type ==
PHANTOM_CPUSYS)))
{
    pthread_mutex_lock(&phantom_pth_ready_mutex);

    pthread_cond_signal(&signal);

    pthread_mutex_unlock(&phantom_pth_ready_mutex);
    //printf("phantom_Pthreads \n");
}
else if (((phantom_compx[phantom_cmpid]->ext_api == phan-
tom_OpenMP))&&((phantom_compx[phantom_cmpid]->dev_prc_type ==
PHANTOM_CPUSYS))
{
    phantom_omp_thread_ready=1;
    #pragma omp flush(phantom_omp_thread_ready)
    printf("phantom OpenMP #pragma omp critical \n");
}
else if (((phantom_compx[phantom_cmpid]->ext_api == phan-
tom_MPI))&&((phantom_compx[phantom_cmpid]->dev_prc_type ==
PHANTOM_CPUSYS)))
{
    MPI_Send(&signal, 1, MPI_UNSIGNED_SHORT, phantom_commobjx[i]->trgt_id, i,
MPI_COMM_WORLD);
}

```

Listing 3-6: First representation of PHANTOM barrier function

3.5 PHANTOM API FOR CPU-GPU COMMUNICATION

This API is focused on the communication between host-CPU devices and the attached GPUs, which are triggered and supplied with data from the host machine, operate in their own environment, and return the required results back to the host machine. The GPU operation cannot be interrupted by the host machine before it fulfills its operation, thus its design includes some different parts from the aforementioned APIs, as further described in the following paragraphs.

3.5.1 Design Specifications

The design of the CPU to GPU communication is inspired by existing GPU tools, such as CUDA and OpenCL (see D4.1). In the context of PHANTOM, the programmer may annotate a function of the component's source code, representing a component to be executed inside a GPU, similar to the following figure:

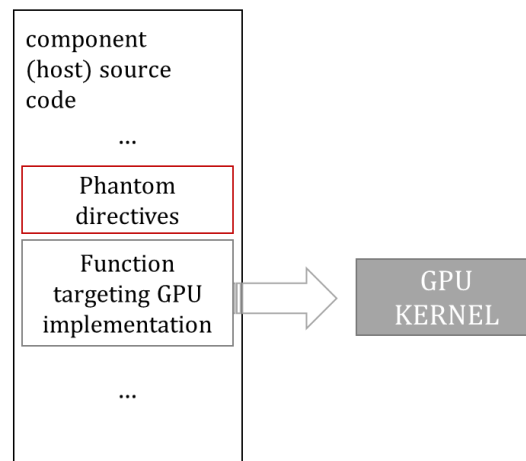


Figure 3-1: High level representation of PHANTOM CPU-GPU API

The annotations are specific PHANTOM pragma directives that include information for the function that will be executed in the GPU device. The function targeting GPU implementation has to be in an appropriate form, aligned with the low-level GPU framework that operates inside the GPU device. The GPU targeted version of the invoked component may be provided by the user or the Parallelization Toolset, which will try to make a GPU version. The Parallelization Toolset and the Deployment Manager are responsible for the selection and replacement of the PHANTOM directives with low-level CPU-GPU communication functions (buffers).

3.5.2 Implementation Details

The PHANTOM CPU-GPU API includes the corresponding directives in the form of PHANTOM pragmas and the low-level implementation which currently consists of CUDA functions targeting NVIDIA GPUs.

The PHANTOM pragma directives are utilized for annotating a function as a GPU targeted component (known also as Kernel), including information for the function that will be executed in the GPU device. The current version of the GPU API consists of two pragmas, one for declaring input and one for declaring output data, to/from the GPU targeted function. The GPU targeted function is specified as phantom kernel:

The PHANTOM pragma for declaring an input to a PHANTOM kernel is represented by the following command:

```
#pragma phantom kernel "function" kernelin "input variable"
    type="variable's type" size="variable's size" (17)
```

This directive declares a GPU targeted function as a *PHANTOM kernel* will have the following attributes:

- *phantom kernel*: the function that is targeted for potential GPU implementation
- *kernelin*: pragma specifier to indicate that the following variable is an input to the gpu function

- *type*: the type of the input variable (needed at least for CUDA API)
- *size*: the size of the input variable (needed at least for CUDA API)

The PHANTOM pragma for declaring an input to a PHANTOM kernel is represented by the following command:

```
#pragma phantom kernel "function" kernelout "output variable" type="variable's type" size="variable's size" (18)
```

This directive declares a GPU targeted function as a *PHANTOM kernel* with the following attributes:

- *phantom kernel*: the function that is targeted for potential GPU implementation
- *kernelout*: pragma specifier to indicate that the following variable is an output of the gpu function
- *type*: the type of the output variable (needed at least for CUDA API)
- *size*: the size of the output variable (needed at least for CUDA API)

These pragmas will be replaced by low-level GPU functions relative to the GPU framework selected by Technique Selection. The current implementation considers the CUDA functions which are exploited using the following procedure:

1. For each GPU pragma declaring an input variable generate appropriate GPU input buffers, as described in the following example:

```
int phantomgpu_size_in_0=64;
double *phantomgpu_dev_in_0=0;
cudaMalloc( (void*)&phantomgpu_dev_in_0,phantomgpu_size_in_0*sizeof(double));
```

Listing 3-7: Example of GPU input variables and functions

2. For each GPU pragma declaring an output variable generate appropriate GPU output buffers, as described in the following example:

```
int phantomgpu_size_out_0=64;
double *phantomgpu_dev_out_0=0;
cudaMalloc( (void*)&phantomgpu_dev_out_0,phantomgpu_size_out_0*sizeof(double));
int phantomgpu_size_in_0=64;
double *phantomgpu_dev_in_0=0;
cudaMalloc( (void*)&phantomgpu_dev_in_0,phantomgpu_size_in_0*sizeof(double));
```

Listing 3-8: Example of GPU output variables and functions

3. Generation of functions that forward the input variables to the GPU in parallel streams:

```
cudaStreamCreate(&stream[phantom_cmpid]);

cudaMemcpyAsync(phantomgpu_dev_in_0, simage, phantomgpu_size_in_0*sizeof(double), cudaMemcpyHostToDevice, stream[phantom_cmpid]);
```

Listing 3-9: Example of variable's forwarding to the GPU device

4. Replacement of the GPU targeted function with its GPU Kernel form:

```
gpufunction_CUDA_kernel <<<dimGrid, dimBlock, 0,  
stream[phantom_cmpid]>>>(simage, simage2, output_image_L, output_image_L2);
```

Listing 3-10: Example of CUDA_kernel function call

5. For each declared output variable generate appropriate GPU functions which retrieve the GPU result:

```
cudaMemcpyAsync(output_image_L, phantomgpu_dev_out_0, phantomg-pu_size_out_0*  
sizeof(double), cudaMemcpyDeviceToHost, stream[phantom_cmpid]);
```

Listing 3-11: Example of GPU functions which retrieve the GPU result

The aforementioned functions enable the triggering of the GPU kernel execution as well as the data exchange between the host CPU machine and the GPU device. The form of the kernel function will be specified by the user, the MOM and the Parallelization Toolset. This API can also be applied to devices, other than GPUs such as FPGAs that are used as accelerators. The implementation of the CPU-GPU API is under progress at the time of the deliverable. Further updates will follow, in the future steps of the development process.

3.6 DEPENDENCIES/INTEGRATION

The Programming Interfaces are strongly related to the Programming Model from where the APIs interact with the corresponding protocols and their variables. Then the Programming Interfaces interact with the Parallelization Toolset and specifically with the Technique Selection which selects the appropriate low-level communication API and replaces specific attributes of the Programming API functions accordingly. In addition, the Programming APIs also interact with the Multi-Objective Mapper, since the selected low-level APIs are essentially derived by the MOM's optimized deployment plan. Furthermore, the Programming Interfaces interact with the Deployment Manager, which generate the low-level communication APIs according to the Technique Selection directives.

3.7 DEMONSTRATOR/EXAMPLE USAGE

The programming APIs are strongly related to the Technique Selection and the Deployment Manager's operation, since they provide important information about the implementation and the generation of the selected APIs.

The following example demonstrates the Programming APIs operation. First, it is assumed that the optimized deployment plan provided by MOM, describes the mapping in the following figure:

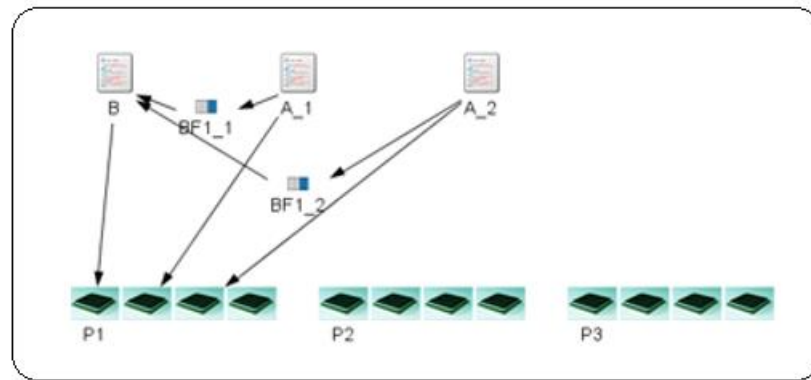


Figure 3-2: Mapping of components in Programming APIs demonstration

The application depicted in the above picture includes three components that communicate with two logical buffers. The C source application includes a Component B which expects data from A_1 , A_2 in order to produce a final summation. Component B waits data from A_1 , A_2 and produces a final summation according to the following listing.

```
int B(int phantom_cmpid ){
...
phantom_wait( &ready_filter, 0);

phantom_synchronize(&output_image_L,phantom_cmpid, 0);
/* PHANTOM Comment: Loop# 1 is-parallel:false */
for (i=0; i<64; i ++ )
{
    final_output+=(((output_image_L[i]*i)/(i+2))+factor);
}
printf("final_output=%f from process_id=%d\n",final_output,phantom_cmpid);
...
}
int A(int phantom_cmpid ){
...
ready_filter=1;
if(phantom_cmpid==2){
phantom_notifyall(&ready_filter,1);
}
phantom_synchronize(&output_image_L,phantom_cmpid, 1);
...
}
```

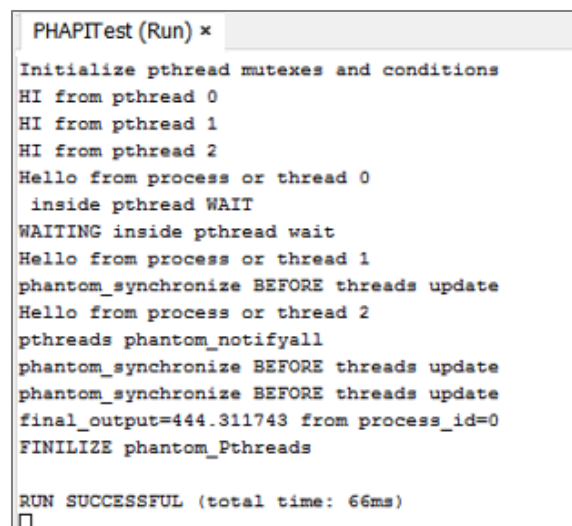
Listing 3-12: Source code of Component B

The components are mapped to the same processor and use two local memories to send and receive data. Technique selection decides (based on processor type (CPU) and communication object type (local memory)) that each component will run in different threads using pthreads. As described in section 2.2.5.1, TS generates the appropriate information in the PHANTOM API header files. Then a preliminary version of the Deployment Manager assigns the identified values to specific C structures from phantom.h and initializes appropriate functionalities relative to pthreads API. Then it creates 3

threads, using `pthread_create()`, one for each component as specified by MOM and indicated by TS. In this example PHANTOM API functions, such as `phantom_synchronize()`, are not replaced by the `pthread` functions. Instead, they activate the corresponding `pthread` functions as described in 3.2.2, 3.3.2, 3.4.2. Finally, the Deployment Manager finalizes the selected low-level communication API. In the case of `pthread`s, the communication API is `pthread`s, thus the Deployment Manager finalizes the operation by destroying mutexes, initiated for thread synchronization.

The demonstrated example is performed under Windows 10, on an Intel i7 processor with 4 cores and 8 threads, with NVIDIA GTX 960M graphics card, and using the NetBeans IDE.

The following figure depicts the console messages provided by the three executed threads:



```

PHAPITest (Run) ×
Initialize pthread mutexes and conditions
HI from pthread 0
HI from pthread 1
HI from pthread 2
Hello from process or thread 0
inside pthread WAIT
WAITING inside pthread wait
Hello from process or thread 1
phantom_synchronize BEFORE threads update
Hello from process or thread 2
pthreads phantom_notifyall
phantom_synchronize BEFORE threads update
phantom_synchronize BEFORE threads update
final_output=444.311743 from process_id=0
FINILIZE phantom_Pthreads

RUN SUCCESSFUL (total time: 66ms)

```

Figure 3-3: Programming APIs example result

As depicted in the above figure, test results show that 3 threads were executed in parallel and successfully synchronized their execution. Specifically, Thread ‘0’ waited for a notification from thread with compid ‘2’, while Threads ‘1’ and ‘2’ performed their execution and filled the appropriate portion of the shared variable. Thread ‘0’ calculated a specified final output based on synchronized shared variable only after it was notified and unblocked.

3.8 INNOVATIONS BEYOND THE STATE-OF-THE-ART

3.8.1 Background technologies utilised in development

PHANTOM programming interfaces do not reuse existing interfaces, but deal with software development of interfaces between interacting components building on top of lower level communication APIs (such as `pthread`s, OpenMP, OpenMPI, CUDA, OpenCL).

3.8.2 Summary of new technologies/extensions developed

The PHANTOM programming model is a superset of the standard C/C++ model, in which components can be written using normal C/C++. The programming model is primarily focused on supporting coordination and data sharing between components.

The programming interfaces assist in the development of component-based applications and include specific directives about parallelization. The provided APIs use the C programming language, enabling the incorporation of parallelization APIs (pthreads, OpenMPI, OpenMP and CUDA also described in D4.1), whilst also providing an abstraction of the system architecture, hiding the complexity between hardware and applications. The programming interfaces are implemented using low-level communication function calls of the selected by Technique Selection (after MOM's decision) hardware-dependent communication APIs. More specifically the low-level APIs that are considered are the pthreads, OpenMP, OpenMPI regarding CPU-CPU communication; and CUDA, regarding CPU-GPU communication.

3.8.3 Early/Full Prototypes functionality

Early-First Year Prototype

The first version of the programming interfaces is focused on the design and implementation of communication API functions for CPU and GPU platforms in relation to code analysis and transformation operations developed in Section 2.3. These programming interfaces have been already developed and tested in standalone proof-of-concepts and not integrated in use case demonstrators for the first-year prototypes.

Full Prototype and Next Steps

The next steps of the Programming Interfaces include modifications and refinements of the low-level communication APIs towards the integration with the PHANTOM Use Cases. Regarding CPU-GPU communication, the current status includes the CUDA API functions, while in the next steps include the incorporation of the OpenCL API. The programming interfaces regarding communication with FPGAs are addressed in WP4 using MPI (section 6.3 in D4.2 and mainly planned for D4.3).

4. MODEL BASED TESTING

Model-Based Testing (MBT) formalizes and automates as many activities related to testing as possible to increase both the efficiency and effectiveness when performing testing tasks. Instead of writing a test case specification with hundreds of pages, MBT automatically generates test cases from MBT models and then executes test cases to test the system under test (SUT).

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 systems. Concretely the following three testing aspects will be defined and implemented within the PHANTOM project.

1) Model validation: During the use case application design phase, MBT creates models following use case specification documents describing the intended implementation of use case applications. The model validation process automatically generates simulation scenarios with necessary inputs to validate the MBT models and highlight unexpected behaviours such as deadlocks or over-designing (parts of the model never activated). This validates the intended implementation of use case applications at a very early design phase without running any tests against SUT.

2) Functional testing: To perform functional testing, MBT automatically generates test cases from MBT models, stimulates applications/components with different test cases on the PHANTOM platform 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 will be reported to developers for checking.

3) Non-functional testing: Non-functional testing tests SUT by use of the same test cases as functional testing but focuses more on performance results or other metrics. At the end of test execution, we compare the monitored values of non-functional attributes (e.g. execution time, energy consumption) with the system requirements to decide if the performance (or power consumption, etc.) of the application or component is satisfactory. In addition to this, non-functional testing is also used to automate processes facilitating the development of PHANTOM platforms (e.g. by providing to the MOM an initial reference mapping between application components and hardware) and use case applications (e.g. by automating the testing process to study the performance influence of application parameters). In addition to non-functional testing for performance, new MBT technology will also be designed and developed to test the correct implementation of PHANTOM security mechanism. This is achieved by simulating the applications to request PHANTOM resources and execute over secured environments; the security implantation in PHANTOM will be tested and validated by comparing the expected and real security reactions.

4.1 USE CASE REQUIREMENTS

The use case requirements that MBT addresses in PHANTOM project is introduced in Table 4-1. Model validation, functional testing and non-functional testing ensure that the correct functioning of applications (U29) and APIs (U30), while the MBT workflow

introduced in section 4.2 along with the MBT open source toolchain section 4.3 provides an API for test implementation (U31).

Table 4-1: Use case requirements addressed by MBT

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

The three use case applications in the right columns (i.e. **Surveillance** use case by GMV, **Telecom** use case by Intecs and **HPC** use case by HLRS) as well as their composing components in PHANTOM are the main SUTs of MBT. Each of the three use cases has identified several functional aspects (input, output, precondition and post condition) to test in terms of specifications for the entire application as well as the composing components; the current available specifications are briefly introduced as follows and will be enriched along with the use case development process.

4.1.1 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.

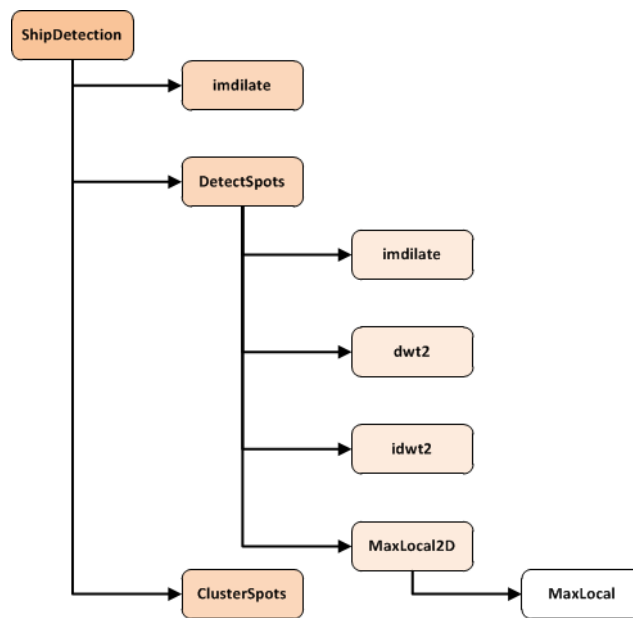


Figure 4-1: ShipDetection component chain

Currently, the new surveillance application defined using PHANTOM programming model is named Ship Detection and has a number of components as illustrated in Figure 4-1; the way how components are composed is also illustrated in the same figure. The functional specification of each component is summarized in Table 4-2.

Table 4-2: Functional specification of Surveillance Use Case

Application/ Component	Inputs	Outputs
ShipDetection	<ul style="list-style-type: none"> - &image - ImageBounds - coastlineSections - Att: extracted metadata from image 	<ul style="list-style-type: none"> - ShipReport
imdilate	<ul style="list-style-type: none"> - &A: matrix of positive integer values - &SE: structuring element object 	<ul style="list-style-type: none"> - &B: matrix of dilated image
DetectSpots	<ul style="list-style-type: none"> - coastlineSections - &image - MaskLand - Miscellaneous 	<ul style="list-style-type: none"> - SpotsT - SpotsD - MiscellaneousOut
dwt2	<ul style="list-style-type: none"> - &image 	<ul style="list-style-type: none"> - &cA: approximation coefficients matrix - &cH: horizontal coefficients matrix - &cV: vertical coefficients matrix - &cD: diagonal coefficients matrix
idwt2	<ul style="list-style-type: none"> - &cA: approximation coefficients matrix - &cH: horizontal coefficients matrix - &cV: vertical coefficients matrix - &cD: diagonal coefficients matrix 	<ul style="list-style-type: none"> - &X: approx. and coefficients matrix
MaxLocal2D	<ul style="list-style-type: none"> - &image - max_spots 	<ul style="list-style-type: none"> - Values: local maximum - row_ind: Row indexes - column_ind: Column indexes
MaxLocal	<ul style="list-style-type: none"> - &image 	<ul style="list-style-type: none"> - Value: array of max values

	<ul style="list-style-type: none"> - dim: matrix dimension to compute - order: selection of sort type - n_loc_max: maximum of local requested 	- max_ind: array of the max indices
ClusterSpots	<ul style="list-style-type: none"> - ShipReportPre - &image - Spots: cluster of input report - Values - LoopIndex: index for storage decisions - Detection: constant configuration 	- SpotsTemp: clustering for input spots

4.1.2 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. Each component runs autonomously on the base of the information from hardware or from shared areas as illustrated in Figure 4-2.

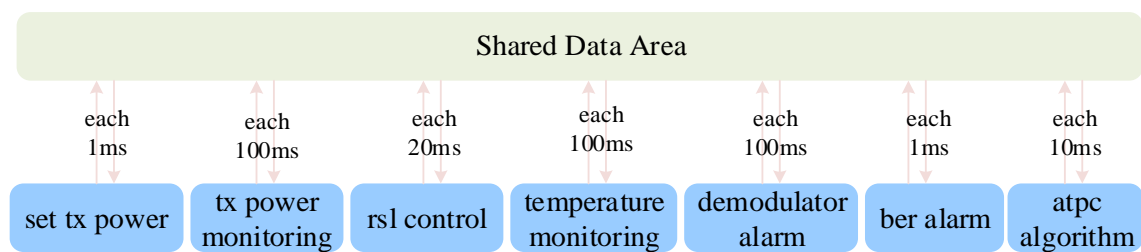


Figure 4-2 Telecom Use Case Architecture

Table 4-3 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 4-3: 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

4.1.3 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.

Figure 4-3 illustrates all components of HPC application and the dependencies, while the available component specification is summarized in Table 4-4 and will be enriched along with the development process.

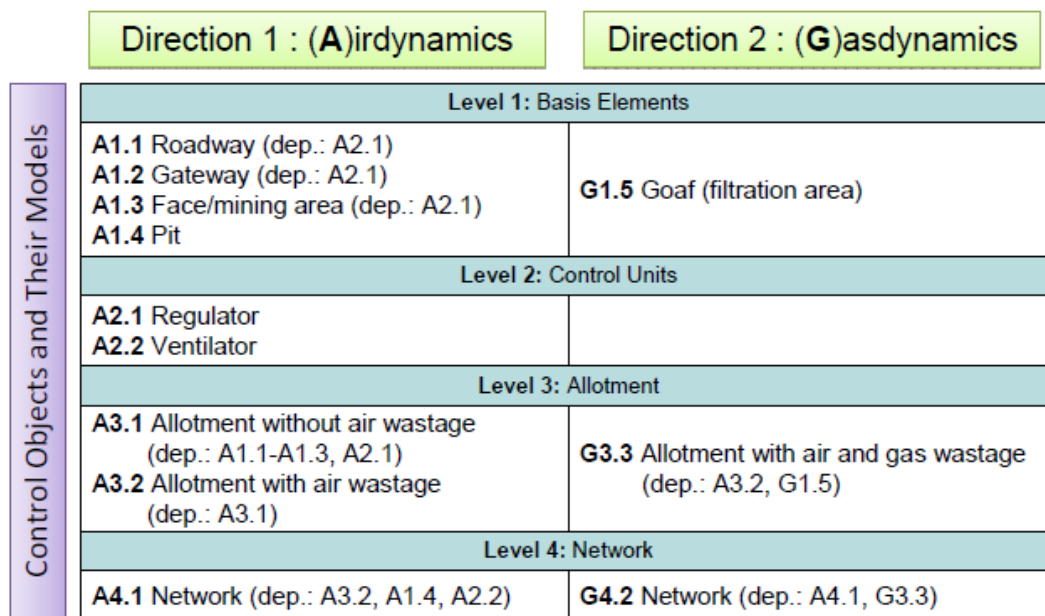


Figure 4-3: HPC application components and dependencies

Table 4-4: Functional specification of HPC Use Case

Application/ Component	Inputs	Output
A1.1 – Roadway	H – the Depression (difference of pressures) at the borders of the element (Pa) r' – the aerodynamic resistance of the local control unit / regulator (Ns ² /m ⁸)	Q – the airflow through the element (m ³ /s)
A3.1 – Allotment without air wastage	H – the Depression (difference of pressures) at the borders of the element (Pa) r' – the aerodynamic resistance of the local control unit / regulator (Ns ² /m ⁸)	Q – the airflow through the element (m ³ /s)

A3.2 – Allotment with air wastage	H – the Depression (difference of pressures) at the borders of the element (Pa) r' – the aerodynamic resistance of the local control unit / regulator (Ns^2/m^8)	Q – the airflow through the roadway and the gateway (m^3/s) Q_s – the airflow through the face (m^3/s) q – the airflow through the goaf / filtration area (m^3/s)
-----------------------------------	---	---

4.2 DESIGN SPECIFICATIONS

Model-based testing is an application of model-based design for generating test cases and executing them against SUT for testing purposes. In PHANTOM, the MBT processes are designed and implemented in four steps as shown in Figure 4-4.

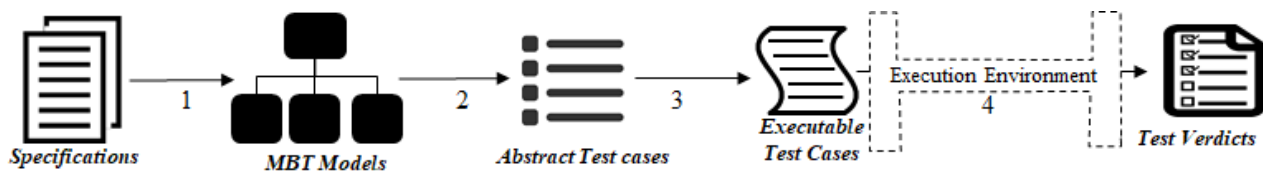


Figure 4-4: PHANTOM MBT workflow

Step 1. Creation of MBT Models. 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 BPMN. In PHANTOM, communicative state machine, as the input of the open source MBT tool DIVERSITY [16], is used for the creation of MBT models. Rationale for these decisions is given in section 4.6.

- Input: use case specification documents
- Output: MBT models

Step 2. Generation of Test Cases. 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). In PHANTOM, we apply different criteria to the same MBT model to generate different sets of test cases for either functional or non-functional testing.

- Input: MBT models, test selection criteria
- 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 by use of codec/decoder 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 SUT.

- Input: abstract test cases, codec/decoder

- Output: concrete test cases

Step 4. Execution of Test Cases. The executable test cases are automatically executed against SUT. To automate test execution, system adapters are required to provide channels connecting SUT with the test execution environment. During the execution, SUT is stimulated by inputs from each test case, and the outputs of SUT are collected to generate test verdicts indicating if a test passes or fails (or is inconclusive). In addition, non-functional information (e.g., execution time, energy consumption) is collected to analyse system performance and facilitate the development process. At last, testing results are reported to developers.

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

The four steps are performed iteratively and incrementally. This means that the MBT activity starts with a basic MBT model containing a few model elements, and generates a small number of test cases to validate a partial implementation of implemented applications. Once the development of applications has further advanced, the MBT models are extended to cover more requirements, and generates more test cases for execution. 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 the very early stage in parallel with the application development and assists the developers through the entire development process.

4.3 IMPLEMENTATION DETAILS

In order to realize the MBT workflow introduced above, MBT components are developed and MBT tools are used in the implementation. Figure 4-5 presents a summary of MBT components developed in PHANTOM (i.e., MBT Models, TTCN-3 Publisher, Codecs/Decoders and System Adapters) and tool support for MBT workflow (i.e., Test Generation Tool and Compilation&Execution Tool).

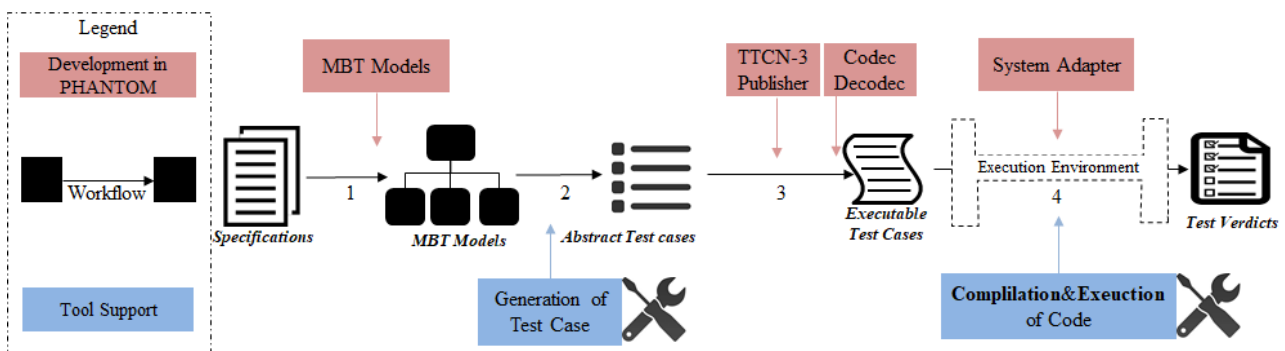


Figure 4-5 MBT Component Development and Tool Support

The functions of MBT components developed in PHANTOM are introduced as follows, while the two tools used to support the MBT workflow, i.e., DIVERSITY [16] and

TITAN [19] respectively for test generation and compilation&execution, are introduced in section 4.6.1.

- **MBT Models:** MBT models are created following application specification and provide an abstraction of structures and behaviours of the intended implementation. They are specific to applications (i.e., telecom, surveillance and HPC) and testing aspects (e.g., functional testing, non-functional testing).
- **TTCN-3 Publisher:** The primary function of TTCN-3 publisher is to transform test cases in other formats (e.g., JUnit, XML) to the standard language TTCN-3; when dealing with test cases generated in TTCN-3 format, TTCN-3 publisher further improves the test cases with bug correction and better modulization.
- **Codecs/Decoders:** Codec/Decoder transforms the abstract test cases to executable ones by providing mappings between the abstraction in MBT models and application implementation details. Codec/Decoder is specific to MBT model.
- **System Adapter:** System adapters support automatic test execution by providing channels for connecting SUT with the test execution environment and data exchange. System Adapter is specific to applications (i.e., telecom, surveillance and HPC) and running environments.

Table 4-5 presents the so far developed MBT components in PHANTOM, in which all components are use case specific except the TTCN-3 publisher applicable for all three use cases.

Table 4-5 MBT components developed in PHANTOM

	<i>Telecom</i>	<i>HPC</i>	<i>Surveillance</i>
MBT Components	MBT Models	MBT Models	MBT Models
	TTCN-3 Publisher		
	Codec/Decoder	Codec/Decoder	Codec/Decoder
	System Adapter for Linux	System Adapter for Linux	System Adapter for Linux
	System Adapter for ZedBoard		

Table 4-6 summarizes the MBT implementation status in PHANTOM regarding the three MBT aspects and four MBT steps introduced above. Generally, MBT components are developed and the entire MBT workflow is realized. Model validation and functional testing have been conducted for the latest version of all three use case applications, while part of non-functional testing has been done for HPC use case.

Table 4-6 MBT Implementation in PHANTOM

	<i>Telecom</i>	<i>HPC</i>	<i>Surveillance</i>
MBT workflow			
<i>Step 1</i>	√	√	√
<i>Step 2</i>	√	√	√
<i>Step 3</i>	√	√	√
<i>Step 4</i>	√	√	√

<i>MBT aspects</i>			
<i>Model Validation</i>	√	√	√
<i>Functional Testing</i>	√	√	√
<i>Non-Functional Testing</i>	...	(√)	...

** for the latest UC applications*

In the following part of this section, we introduce the details of MBT components developed in PHANTOM; in the next section, we introduce the execution demonstration with testing results.

4.3.1 MBT models

Following the specifications (briefly introduced in section 4.1), we have created MBT models for each use case. Generally, the telecom use case application is function-driven in which a number of control flows dominate the application and the MBT models for telecom use case contain a number of states and transitions correspondent with the application control flows; while the surveillance and HPC use cases are data-driven in which applications focus more on data flow to take inputs and generate outputs and thus the MBT models for the two use cases are rather simple with input and output data relations.

MBT models are created based on the metamodel “communicative state machine” in xLIA language (eXecutable Language for Interaction & Assemblage). Communicative state machine is an extension of UML state machine to take into account the communications among internal components and parallel architecture in PHANTOM. The MBT models are created in a textual environment by xLIA language; in order to better present the MBT models, Figure 4-6, Figure 4-7 and Figure 4-8 respectively illustrate the graphical visualization of MBT models for Telecom, Surveillance and HPC use cases. However, the MBT models for Telecom and HPC use cases are too big to present all the details in one page, so in Figure 4-6 and Figure 4-8 we present a global view of the models in the first half of the figure while a zoom view with details in the second half.

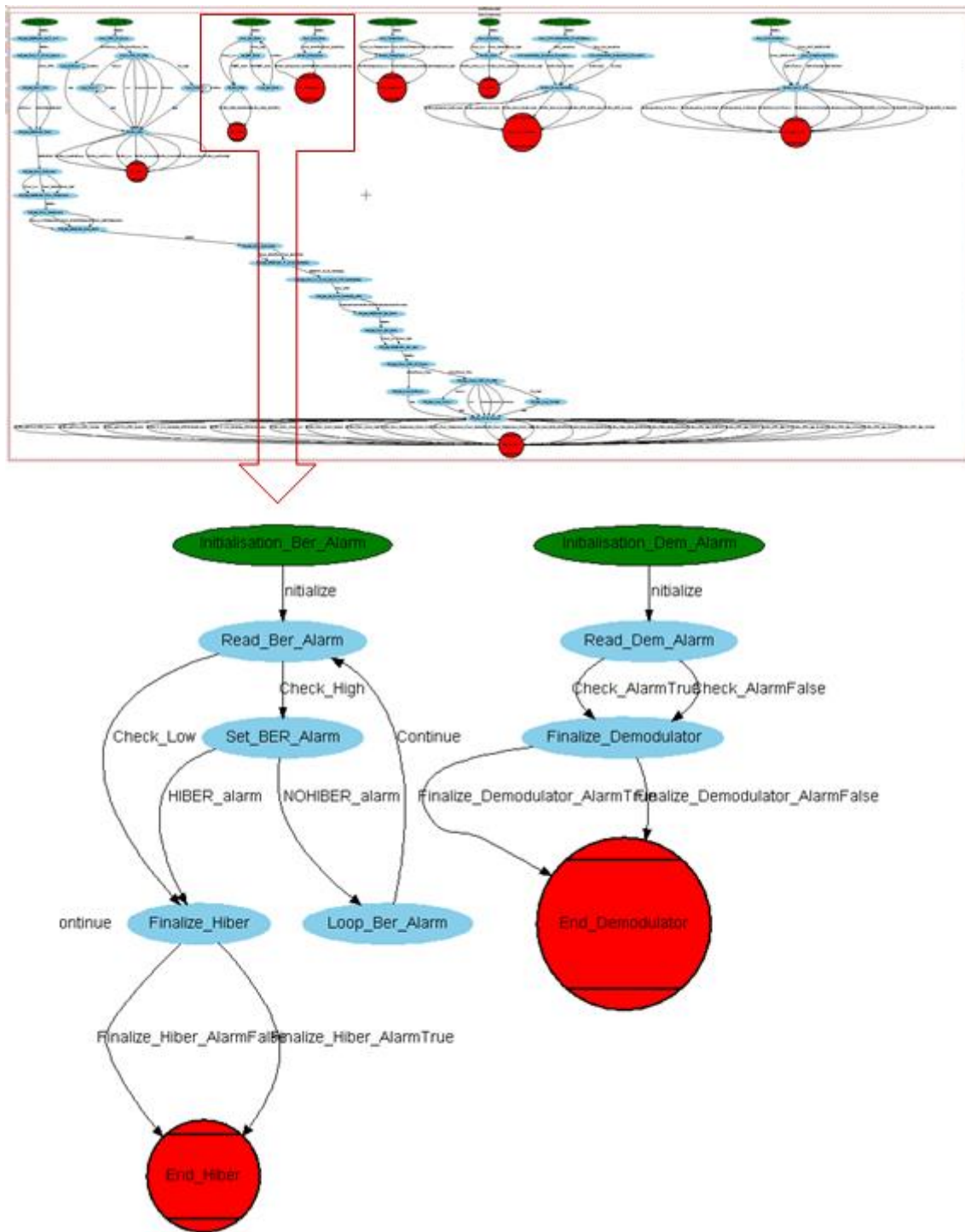


Figure 4-6 MBT models for telecom use case

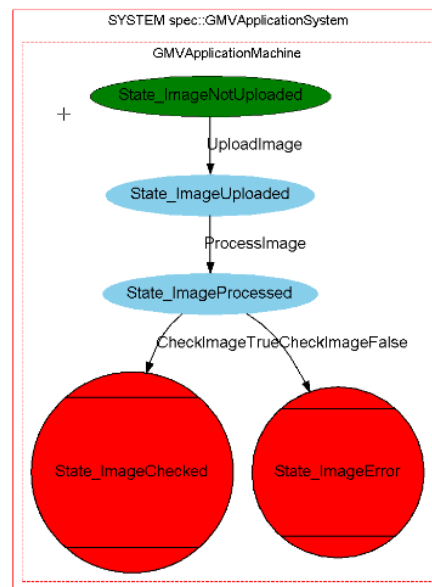


Figure 4-7: MBT models for surveillance use case

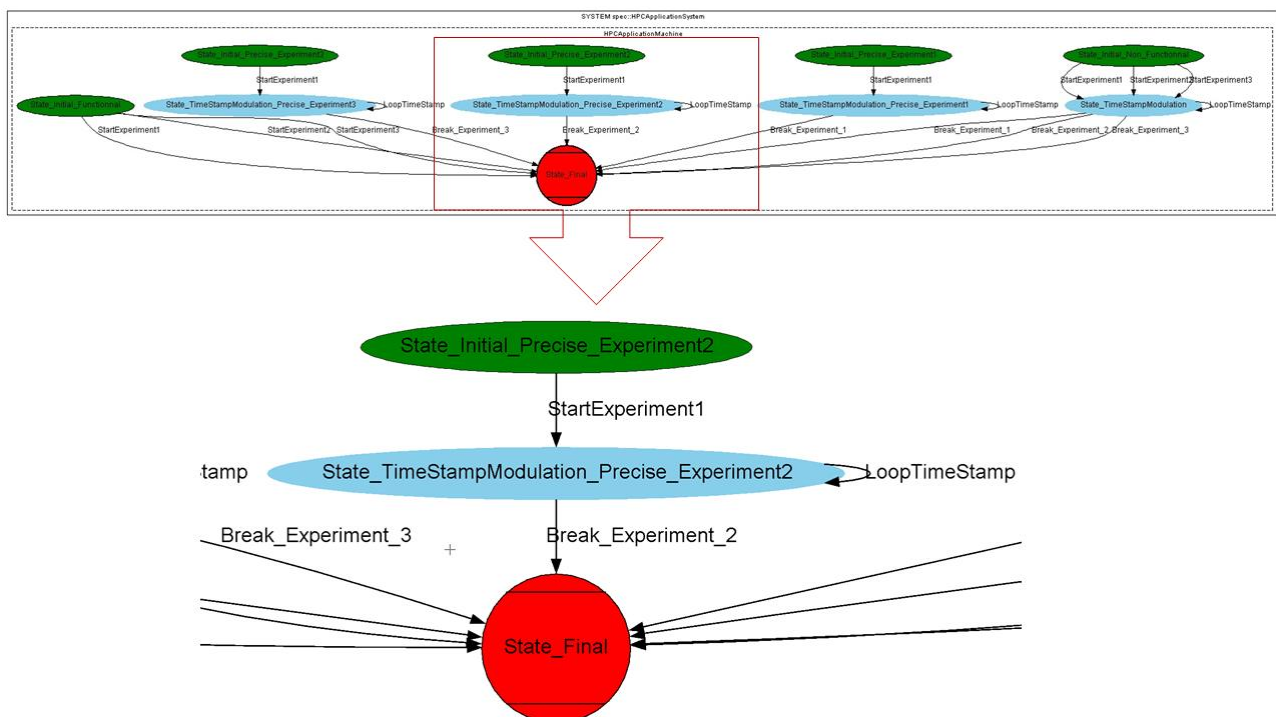


Figure 4-8 MBT models for HPC use case

Current MBT models for telecom and surveillance use cases are function-centric, and MBT model for HPC use case contains both functional testing and non-functional testing. MBT conducts functional testing, non-functional testing as well as model validation for both data-driven and function-driven systems in PHANTOM. The emphasis on non-functional testing is being put on data driven applications i.e., HPC and surveillance use cases, to further investigate other possibilities how MBT facilitates the development process by providing execution related non-functional information with developers. This aspect has been identified and is presented in section 4.6.

4.3.2 Generated Test cases

The test cases in TTCN-3 are then generated from the test generation tool DIVERSITY by applying selection criteria to MBT models. At the current stage, we model specifications as 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. Figure 4-9 shows the extraction of the generated telecom test cases (top left), HPC test cases (top right) and surveillance test cases (bottom). Figure 4-7 summarizes the test case generation results.

```

module TTCN_TestsAndControl {

  import from TTCN_Declarations all;
  import from TTCN_Templates all;

  altstep RTDS_fail() runs on SSM {
  }

  testcase TC_tracel() runs on runsOn_SSM system SSM {
    activate(RTDS_fail());
    cEnv.send(SetupStartingConfiguration_tracel_LINK_0)
    cEnv.send(SetupStepRegisters_tracel_LINK_1)
    cEnv.send(Execute_tracel_LINK_2)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_3)
    cEnv.send(RegisterAndAlarmValues_tracel_LINK_4)
    cEnv.receive(CheckRegisterAndAlarmValues_tracel_LINK_5)
    setverdict(pass)
  }

  testcase TC_trace2() runs on runsOn_SSM system SSM {
    activate(RTDS_fail());
    cEnv.send(SetupStartingConfiguration_trace2_LINK_0)
    cEnv.send(SetupStepRegisters_trace2_LINK_1)
    cEnv.send(Execute_trace2_LINK_2)
    cEnv.receive(ResponseExecutionResult_trace2_LINK_3)
    cEnv.send(RegisterAndAlarmValues_trace2_LINK_4)
    cEnv.receive(CheckRegisterAndAlarmValues_trace2_LINK_5)
    setverdict(pass)
  }
}

module TTCN_TestsAndControl {

  import from TTCN_Declarations all;
  import from TTCN_Templates all;

  altstep RTDS_fail() runs on HPCApplicationSystem {
  }

  testcase TC_tracel() runs on runsOn_HPCApplicationSystem system HPCApplicationSystem {
    activate(RTDS_fail());
    cEnv.send(RequestExecute_tracel_LINK_0)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_1)
    cEnv.send(RequestExecute_tracel_LINK_2)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_3)
    cEnv.send(RequestExecute_tracel_LINK_4)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_5)
    cEnv.send(RequestExecute_tracel_LINK_6)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_7)
    cEnv.send(RequestExecute_tracel_LINK_8)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_9)
    cEnv.send(RequestExecute_tracel_LINK_10)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_11)
    cEnv.send(RequestExecute_tracel_LINK_12)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_13)
    cEnv.send(RequestExecute_tracel_LINK_14)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_15)
    cEnv.send(RequestExecute_tracel_LINK_16)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_17)
    cEnv.send(RequestExecute_tracel_LINK_18)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_19)
    cEnv.send(RequestExecute_tracel_LINK_20)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_21)
    cEnv.send(RequestExecute_tracel_LINK_22)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_23)
    cEnv.send(RequestExecute_tracel_LINK_24)
  }
}

module TTCN_TestsAndControl {

  import from TTCN_Declarations all;
  import from TTCN_Templates all;

  altstep RTDS_fail() runs on GMVApplicationSystem {
  }

  testcase TC_tracel() runs on runsOn_GMVApplicationSystem system GMVApplicationSystem {
    activate(RTDS_fail());
    cEnv.send(RequestUpload_tracel_LINK_0)
    cEnv.send(RequestExecute_tracel_LINK_1)
    cEnv.receive(ResponseExecutionResult_tracel_LINK_2)
    cEnv.receive(ResponseProcessedImage_tracel_LINK_3)
    setverdict(pass)
  }
}

```

Figure 4-9: Test cases generated from telecom, surveillance and HPC models

Table 4-7: Summary of test case generation

	Telecom	Surveillance	HPC
Number of state machine	8	1	1
Number of test cases	60	1	9
Transition coverage	100%	100%	100%

The table rows illustrate, for each use case, the number of state machines created in the MBT models, the number of test cases generated from models and the transition coverage for each MBT model. 100% transition coverage indicates that all current testing requirements defined in the specification documents are covered by the

generated test cases and the execution of the tests will be able to test all expected aspects. Currently surveillance use case has only 1 test case, because it does not yet support the execution of individual component but only entire application; on the other hand, the output of each component from surveillance use case is recorded during the application execution, and thus MBT is able to take into account the outputs of all components and application with only one test case and test both components and use case application. The MBT models and the test cases will continue to be enriched along with the application development and the enrichment of non-functional testing strategies.

As an open source tool for test generation, DIVERISTY is powerful for its generation algorithm, i.e., symbolic execution [18], but also comes with some limitations (e.g., lack of documentation). Particularly, the TTCN-3 format that DIVERSITY used doesn't fully align with the ETSI (European Telecommunications Standards Institute) standard specification of TTCN-3, and the generated test cases contain grammatical errors. Thus, the test generation is further assisted by the MBT component TTCN-3 publisher for error correction and better modulization, which is introduced as follows.

4.3.3 TTCN-3 Publisher

TTCN-3 Publisher is a MBT component developed in PHANTOM in Java for all use cases. The primary function of TTCN-3 publisher is to transform test cases in other formats (e.g., JUnit, Perl) to standard testing language TTCN-3 [17]; when dealing with test cases already generated in TTCN-3 format, TTCN-3 publisher further improves the test cases with bug correction and better modulization.

TTCN-3 publisher takes test cases generated from DIVERISTY as input, corrects grammatical errors and further improves the modulization of test cases to better manage test complexity. For illustration purpose, Figure 4-10 presents the development interface of TTCN-3 Publisher; Figure 4-11 shows an extraction of test cases generated by DIVERSITY and Figure 4-12 shows an extraction of test cases improved by TTCN-3 Publisher with bug correction, details of application abstraction, and modulization.

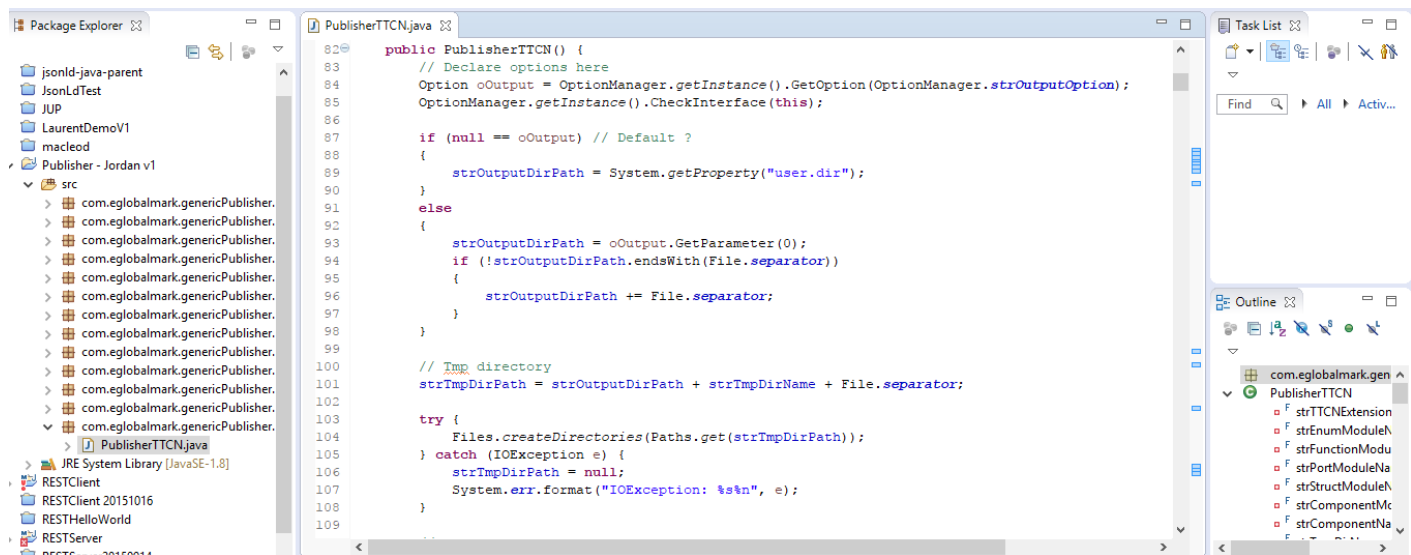


Figure 4-10 TTCN-3 Publisher Implementation

```

module TTCN_TestsAndControl {

    import from TTCN_Declarations all;
    import from TTCN_Templates all;

    altstep RTDS_fail() runs on SSM {
    }

    testcase TC_trace1() runs on runsOn_SSM system SSM {
        activate(RTDS_fail());
        cEnv.send(SetupStartingConfiguration_trace1_LINK_0)
        cEnv.send(SetupStepRegisters_trace1_LINK_1)
        cEnv.send(Execute_trace1_LINK_2)
        cEnv.receive(ResponseExecutionResult_trace1_LINK_3)
        cEnv.send(RegisterAndAlarmValues_trace1_LINK_4)
        cEnv.receive(CheckRegisterAndAlarmValues_trace1_LINK_5)
        setverdict(pass)
    }

    testcase TC_trace2() runs on runsOn_SSM system SSM {
        activate(RTDS_fail());
        cEnv.send(SetupStartingConfiguration_trace2_LINK_0)
        cEnv.send(SetupStepRegisters_trace2_LINK_1)
        cEnv.send(Execute_trace2_LINK_2)
        cEnv.receive(ResponseExecutionResult_trace2_LINK_3)
        cEnv.send(RegisterAndAlarmValues_trace2_LINK_4)
        cEnv.receive(CheckRegisterAndAlarmValues_trace2_LINK_5)
        setverdict(pass)
    }
}

```

Figure 4-11 Extraction of Test Cases Generated by DIVERSITY

```

module TTCN_TestSuite {

    import from TTCN_Enumerations all;
    import from TTCN_Functions all;
    import from TTCN_Ports all;
    import from TTCN_Structures all;
    import from TTCN_Component all;

    import from Intecs_Functions all; // TODO : fix preamble and postambul in TTCN_Functions

    testcase TestCase_1() runs on TestComponent {
        f_preamble();
        fSetupStartingConfiguration({ E_COMP_SET_TX_POWER, 0, E_TPC_MODE_SQUELCHED, E_TPC_MODE_MAINTENANCE, E_NO_DEBUG, -17});
        fSetupStepRegisters({ 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, -18});
        fExecute(INTECS_APPLICATION);
        fResponseExecutionResult(NO_ERROR);
        fRegisterAndAlarmValues({ { 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -96}, false, false, false, false, false, false, false, false, 0, 0, E_CONSTANT});
        fCheckRegisterAndAlarmValues(true);
        f_postambul();
        setverdict(pass);
    }

    testcase TestCase_2() runs on TestComponent {
        f_preamble();
        fSetupStartingConfiguration({ E_COMP_SET_TX_POWER, 0, E_TPC_MODE_SQUELCHED, E_TPC_MODE_MAINTENANCE, E_NO_DEBUG, 480});
        fSetupStepRegisters({ 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0});
        fExecute(INTECS_APPLICATION);
        fResponseExecutionResult(NO_ERROR);
        fRegisterAndAlarmValues({ { 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -96}, false, false, false, false, false, false, false, false, 0, 0, E_CONSTANT});
        fCheckRegisterAndAlarmValues(true);
        f_postambul();
        setverdict(pass);
    }
}

```

Figure 4-12 Extraction of Test Cases Improved by TTCN-3 Publisher

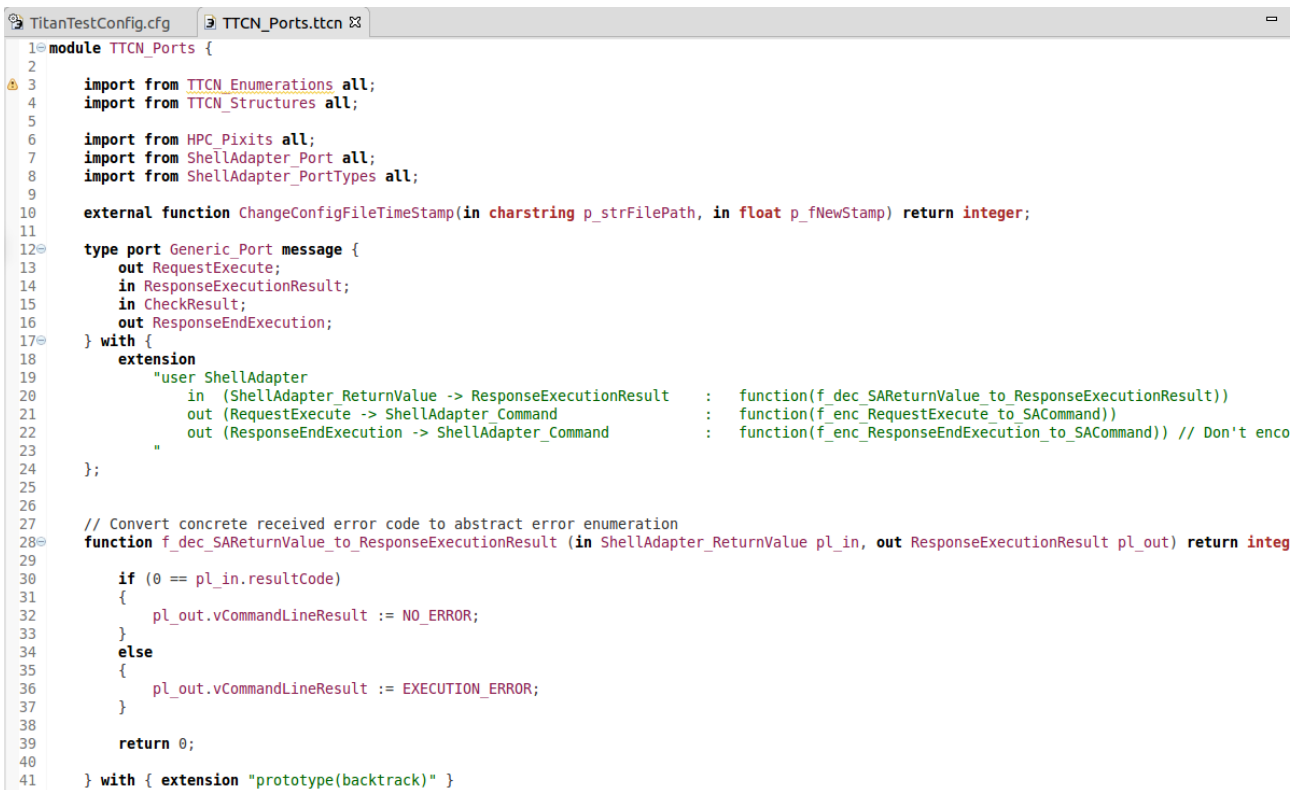
4.3.4 Codec/Decodec

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. Codec/Decodec is specific to MBT model and the relative generated test cases.

The following three Codecs/Decodecs have been developed in TTCN-3 so far in PHANTOM to support the concretization of test cases for three use case applications.

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

For illustration purpose, Figure 4-13 presents an extraction of the TTCN-3 code of Codec/Decodec for HPC application test cases.



```

1 module TTN_Ports {
2
3   import from TTCN_Enumerations all;
4   import from TTCN_Structures all;
5
6   import from HPC_Pixits all;
7   import from ShellAdapter_Port all;
8   import from ShellAdapter_PortTypes all;
9
10  external function ChangeConfigFileTimeStamp(in charstring p_strFilePath, in float p_fNewStamp) return integer;
11
12  type port Generic_Port message {
13    out RequestExecute;
14    in ResponseExecutionResult;
15    in CheckResult;
16    out ResponseEndExecution;
17  } with {
18    extension
19      "user ShellAdapter
20        in (ShellAdapter_ReturnValue -> ResponseExecutionResult) : function(f_dec_SAReturnValue_to_ResponseExecutionResult))
21        out (RequestExecute -> ShellAdapter_Command) : function(f_enc_RequestExecute_to_SACCommand))
22        out (ResponseEndExecution -> ShellAdapter_Command) : function(f_enc_ResponseEndExecution_to_SACCommand)) // Don't enco
23      "
24  };
25
26
27  // Convert concrete received error code to abstract error enumeration
28  function f_dec_SAReturnValue_to_ResponseExecutionResult (in ShellAdapter_ReturnValue pl_in, out ResponseExecutionResult pl_out) return integ
29
30    if (0 == pl_in.resultCode)
31    {
32      pl_out.vCommandLineResult := NO_ERROR;
33    }
34    else
35    {
36      pl_out.vCommandLineResult := EXECUTION_ERROR;
37    }
38
39    return 0;
40
41  } with { extension "prototype(backtrack)" }

```

Figure 4-13 TTCN-3 Code Extraction for HPC Codec/Decodec

4.3.5 System Adapter

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 project, PHANTOM platform will eventually

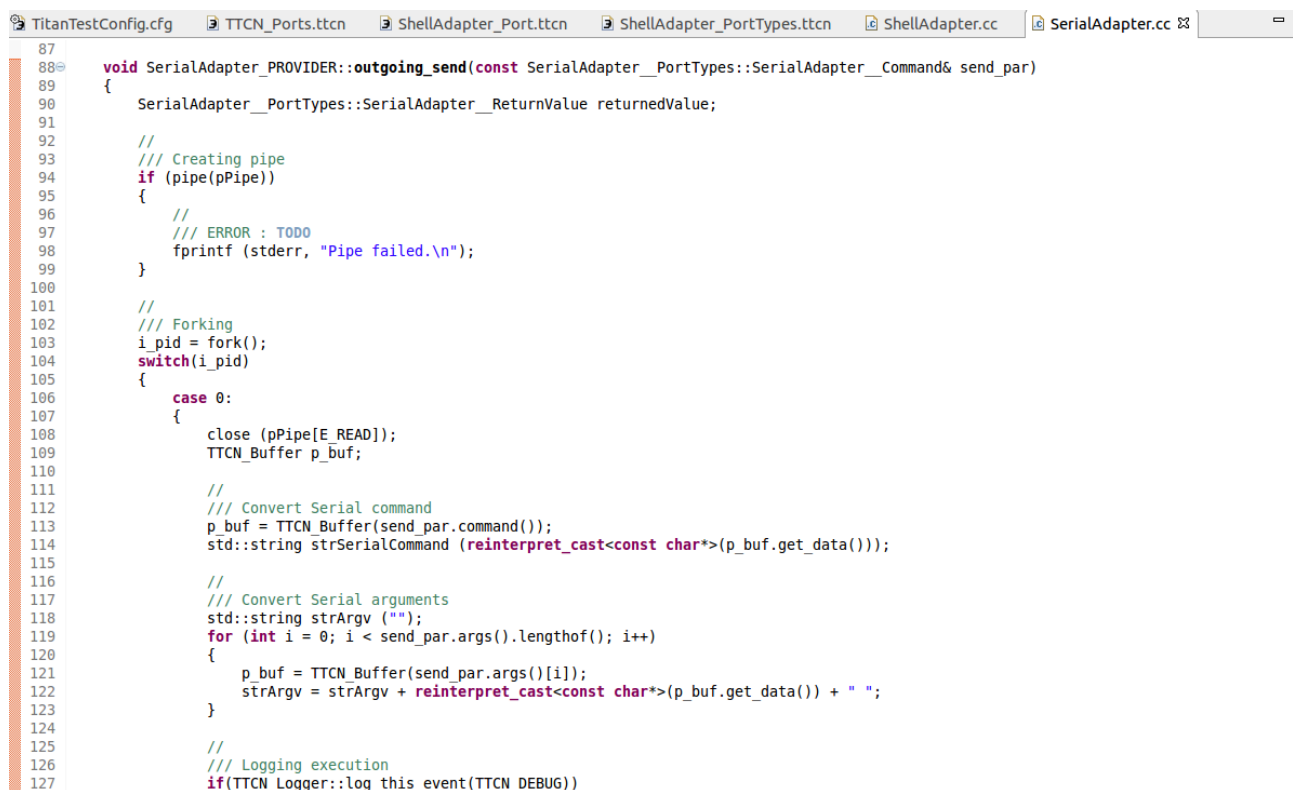
provide an abstraction over different applications and running environments, and thus a unified application-agnostic and environment-agnostic interface for users and testers.

For the time being, since the PHANTOM interface for use case applications is not fully available yet, the following four system adapters have been developed in TTCN-3 and C++ to enable early test of use case applications and 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

Once the PHANTOM interface for use case applications is available, the MBT system adapter for PHANTOM interface will be developed to test applications integrated in PHANTOM platform over distributed computing environments.

For illustration purpose, Figure 4-13 presents the code extraction of the system adapter for telecom application over ZedBoard.



```

87
88 void SerialAdapter_PROVIDER::outgoing_send(const SerialAdapter_PortTypes::SerialAdapter_Command& send_par)
89 {
90     SerialAdapter_PortTypes::SerialAdapter_ReturnValue returnedValue;
91
92     //
93     /// Creating pipe
94     if (pipe(pPipe))
95     {
96         //
97         /// ERROR : TODO
98         fprintf(stderr, "Pipe failed.\n");
99     }
100
101     //
102     /// Forking
103     i_pid = fork();
104     switch(i_pid)
105     {
106     case 0:
107     {
108         close (pPipe[E_READ]);
109         TTCN_Buffer p_buf;
110
111         //
112         /// Convert Serial command
113         p_buf = TTCN_Buffer(send_par.command());
114         std::string strSerialCommand (reinterpret_cast<const char*>(p_buf.get_data()));
115
116         //
117         /// Convert Serial arguments
118         std::string strArgv ("");
119         for (int i = 0; i < send_par.args().lengthof(); i++)
120         {
121             p_buf = TTCN_Buffer(send_par.args()[i]);
122             strArgv = strArgv + reinterpret_cast<const char*>(p_buf.get_data()) + " ";
123         }
124
125         //
126         /// Logging execution
127         if(TTCN_Logger::log_this_event(TTCN_DEBUG))

```

Figure 4-14 Code Extraction for Telecom System Adapter over ZedBoard

4.3.1 Implementation Summary

Regarding the latest applications of three use cases introduced in section 4.1, the end-to-end MBT process has been implemented:

- 1) MBT models have been created respectively for three use case applications;
- 2) Test cases have been generated covering 100% of the expected testing aspects;
- 3) TTCN-3 publisher has been developed to improve the generated test cases;
- 4) Codecs/Decoders have been developed to concretize test cases;
- 5) System adapters have been developed to enable automatic test execution;
- 6) Test cases have been executed and testing results have been provided to use case partners.

The next section presents the execution demonstration with testing results.

4.4 DEMONSTRATION AND TESTING RESULTS

MBT has been conducted for the latest version of three use applications with the following three aspects:

- **Model validation:** early model validation indicates that the intended implementations of the three use case applications do not contain deadlock nor over design.
- **Functional Testing:** functional testing has found different implementation defects of the three use case applications, which can be broadly divided into two categories: a) functions are not correctly implemented, and b) application interfaces (e.g., data formats) are not aligned with specification. Functional testing results have been sent to partners and the corresponding debugs are over (for HPC and surveillance use cases) or ongoing (for telecom use case).
- **Non-functional testing:** Non-functional testing has been so far conducted on HPC use case to study the relation between parameter value and simulation steps. This progress is automated and accelerated by MBT in the way that MBT model generates test cases with different values of the same parameter, executes the application a number of times and collects simulations results on parameter values. Non-functional testing will be the main focus point MBT during the project's third year including security testing and performance testing with monitoring framework and MOM.

In addition to this, MBT for PHANTOM security is being designed to test the security implementation in PHANTOM; this is achieved by simulating the application activities to request resources and execute over secured environment, which will be another main focus of MBT work in the third year.

For illustration purpose, Figure 4-15, Figure 4-16 and Figure 4-17 respectively illustrate the model validation result for surveillance application, the functional testing results for telecom application and non-functional results for HPC application.

```

REPORT
STOP CRITERIA PROCESSOR
The CONTEXT count : 5
The STEP count : 3

The Max HEIGHT reaching : 2
The Max WIDTH reaching : 1
The RUN#EXIT count : 2
TRANSITION COVERAGE PROCESSOR
All the << 4 >> transitions are covered !
Number of nodes cut back: 0

step: 0 , context:+oo , height:+oo , width:+oo
stop: 0 , context: 5 , height: 1 , width: 1
--> 0ns @ Wed Nov 01 22:14:38 2017

REPORT
STOP CRITERIA PROCESSOR
The CONTEXT count : 5
The STEP count : 0

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

```

Figure 4-15 Model Validation Result for GMV application

```

MTC@artemis-EGM: Stopping test component execution.
MTC@artemis-EGM: Test case TestCase_60 was stopped.
MTC@artemis-EGM: Terminating component type TTCN_Component.TestComponent.
MTC@artemis-EGM: Default with id 1 (altstep RTDS_fail) was deactivated.
MTC@artemis-EGM: Port cPortGeneric_Port was stopped.
MTC@artemis-EGM: Component type TTCN_Component.TestComponent was shut down inside
testcase TestCase_60.
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: fail reason: ""TestCase_60: Unexpected me
ssage received""
MTC@artemis-EGM: No PTCs were created.
MTC@artemis-EGM: Test case TestCase_60 finished. Verdict: fail reason: "TestCase
_60: Unexpected message received"
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 %), 46 pass (76.67 %), 0 incon
c (0.00 %), 14 fail (23.33 %), 0 error (0.00 %).
MTC@artemis-EGM: Test execution summary: 60 test cases were executed. Overall ve
rdict: 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 4-16 Functional Testing Result of Telecom application

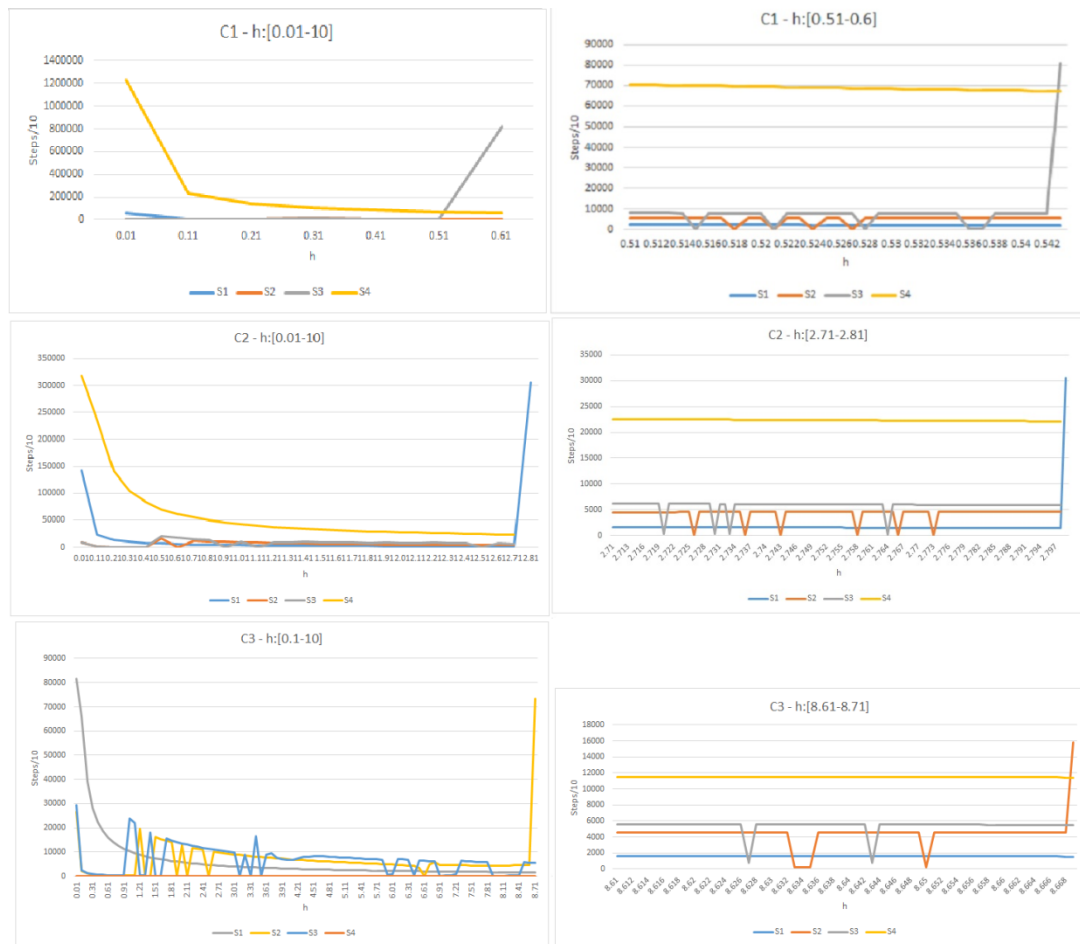


Figure 4-17 Non-functional Testing Result of HPC application

From the three figures above we can see that,

- 1) model validation of GMV application indicates no overdesign or deadlock in the intended implementation;
- 2) 46/60 test cases pass for functional testing on telecom use case. For the failed tests, MBT keeps traces to associate elements from specifications to MBT models and then to test cases for later debug.
- 3) Non-functional testing results of HPC application contain 6 diagrams on three physical configurations (i.e., C1, C2 and C3) with total 600 simulations with different input parameter values. The x axis represents the value of the parameter h ; the y axis represents the total simulation steps when given different h value. The two diagrams on top, in the middle and at the bottom respectively relate to C1, C2 and C3. The three diagrams in the left side provides a normal granularity with one hundred h values from 0.01 to 10, and the three diagrams in the right side provides a refined granularity with one hundred h values for each configuration.

4.5 DEPENDENCIES/INTEGRATION

In addition to developing model based testing solutions, progress has also been made in integration between MBT and the PHANTOM platform. This section introduces the integration objectives to address, the identified PHANTOM interfaces and the MBT interaction flow with PHANTOM platform when executing tests.

4.5.1 Integration objectives

The three following objectives are identified as the key points for integration at the current stage, and the objectives are addressed in the integration plan in the following sections.

Objective 1: To identify the interactions between the MBT workflow and the PHANTOM architecture modules (i.e. MOM, Deployment Manager, Monitoring Framework and Repository).

MBT calls the Deployment Manager to conduct tests on different HW, get functional and non-functional information from the repository and monitoring framework, and provide testing results for the MOM as references. It is thus necessary to identify the interaction patterns between MBT and PHANTOM modules.

Objective 2: To define the interfaces to communicate with use case applications in PHANTOM.

Use case developers intend to use PHANTOM native interfaces to send input to application and get back output data. It is useful for MBT to identify PHANTOM interfaces for use case applications in PHANTOM for testing purposes.

Objective 3: To identify the deployment plan to test both an application and the components from which it is composed.

When an application is deployed in the PHANTOM platform, MBT needs to test both the entire application and its individual components. Therefore, it is necessary to identify the deployment plans in PHANTOM to execute either an application or only one individual component.

4.5.2 PHANTOM platform interfaces

The PHANTOM platform will have one REST server as the first Repository interface (among others) for both testing and real applications, which allows developers, testers and users to send and receive necessary information for task execution. The necessary information sent and received via REST server is stored in the PHANTOM repository and includes (but is not limited to):

- The Platform Description, which describes all deployed HW in PHANTOM platform;
- The Component Network, which describes all software components of an application and the ways how the components are connected;

- The Deployment Plan, which describes the execution mapping between the Platform Description and Component Network;
- Input/Output data;
- Start Trigger and end signal of execution.

MBT can be in charge of manually starting the server by running a script, or PHANTOM can keep the server running when the platform is started.

In addition to REST interfaces, two other interfaces a) local/cloud file storage, and b) TCP/IP for local connections are also considered in PHANTOM.

4.5.3 MBT interaction flow with PHANTOM

The general interaction flow between MBT and the PHANTOM platform is identified as follows, and is used as a reference to guide the process when MBT executes tests on PHANTOM platform.

Step 1. MBT firstly gets from REST sever a) PlatformDescription.xml, and b) ComponentNetwork.xml.

Step 2. MBT sends ComponentNetwork.xml and DeploymentPlan.xml to the REST server. Sending ComponentNetwork.xml is optional, and it is only necessary when MBT tests individual component instead of the whole application: in this case, the ComponentNetwork.xml contains only description about one component and PHANTOM will use the new received ComponentNetwork.xml to deploy components to be executed. Sending DeploymentPlan.xml is also optional, and it is only necessary when MBT force the mapping between a component task and a certain hardware: in this case, the MOM takes into account the received DeploymentPlan.xml when mapping tasks to HW.

Step 3. MBT sends input data to the REST server.

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

Step 5. Once the execution is over, the platform sends back to a) end signal of execution, and b) output data. 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 6. MBT gets non-functional information from the REST server. This step is necessary when non-functional testing is performed.

Step 7. MBT sends non-functional testing results to the REST server in order to be considered by the MOM. This step is optional and it is only necessary at the early stage when the MOM requires an initial mapping reference from MBT regarding the performance information between computing tasks and HW.

4.6 INNOVATIONS BEYOND THE STATE-OF-THE-ART (EGM)

Based on the design and implementation of MBT in PHANTOM, EGM is under the preparation of an innovation patent. Within the PHANTOM project, model validation and MBT component development are in parallel with the development of use case applications to enable early validation intended implementations and later testing, while test execution is conducted right after the applications are ready to execute.

4.6.1 Background technologies utilised in development

In addition to the MBT components developed in PHANTOM, two tools are also used to achieve the MBT workflow, i.e. DIVERSITY for test generation (step 2) and TITAN for test compilation and execution (step 4).

DIVERSITY [16] is an open-source Eclipse based tool for formal analysis. It takes FSM models defined in xLIA (eXecutable Language for Interaction & Assemblage) [16] as input and generates test cases in TTCN-3 [17] following three selection criteria (i.e. exploration, transition coverage and behaviour selection). TTCN-3 is a standardized testing language with multiple testing purpose support (e.g. real-time support, distributed testing support) developed by ETSI (European Telecommunication Standards Institute). Furthermore, symbolic execution algorithm [18] is used by DIVERSITY to use symbolic parameters for inputs rather than numerical values to generate multiple test cases at the same time and more efficiently explore available test cases. DIVERSITY also provides functionality for MBT model validation to detect unexpected behaviours of SUT in the design time. However, as mentioned before, the TTCN-3 format that DIVERSITY uses doesn't fully align with the ETSI standard specification, and the generated test cases contain grammatical errors. Thus, it is necessary to develop the TTCN-3 publisher to improve the test cases generated by DIVERSITY and support the test execution in PHANTOM.

TITAN [19] is a TTCN-3 compilation and execution environment with an Eclipse-based IDE supporting both functional and non-functional testing as well as result reporting. In design time, it is able to generate C++ code from TTCN-3 for further test development and instrument the generated C++ code for future execution; in the runtime, it keeps up running the SUT, in case of a runtime error, TITAN runtime control cleans up the test system, assigns an "error" verdict to the given test case and starts execution of the next test case. Together with TITAN, a toolbox containing support for numerous protocols and several system adapters are also released to open source.

In PHANTOM, we chose DIVERSITY and TITAN to support MBT development due to three main reasons as follows:

- Open source tool: both DIVERSITY and TITAN are open source tools with high flexibility for test design and maintaining, free and easy to use, security and potential customizability for extensions.
- Support of TTCN-3 based test cases: as a global standard testing language, TTCN-3 has the advantage 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 PHANTOM distributed embedded environment.

- **Symbolic execution:** DIVERSITY uses symbolic execution to facilitate the MBT modelling and test generation process while improving the MBT efficiency and performance.

In the current phase, DIVERSITY and Titan covers most of our testing requirements in PHANTOM; on the other hand, some disadvantages (e.g. lack of graphical interface and documentation) in DIVERSITY still exist. Whilst developing MBT solutions by use of DIVERSITY and TITAN, we are also exploring other potential MBT tools (preferably open source ones) to extend MBT activities in PHANTOM.

4.6.2 Summary of new technologies/extensions developed

As illustrated in Table 4-5, 11 MBT components have been developed in PHANTOM for both functional and non-functional testing of use case applications, including:

- **MBT 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 take into account parallel architecture and communication between components of PHANTOM applications.
- **TTCN-3 Publisher** in *Java* for all test cases. The TTCN-3 publisher provides functions of format conversion, bug fix and better modulization to better manage test complexity in PHANTOM.
- **Codecs/Decoders** in *TTCN-3* for all three uses cases.
- **System Adapters** in *TTCN-3* and *C++* for all three use cases on Linux as well as another system adapter for telecom use case on ZedBoard.

The MBT components developed in PHANTOM are reusable for further test of specific applications and will also be updated and enriched along with the development and integration activities of PHANTOM.

Based on the components developed in PHANTOM, model validation is designed and conducted to test the intended implementations in an early stage before executing applications. Among functional and non-functional testing for use cases, non-functional testing for parameter study in HLRS application is an extension since the MBT does not only generate the test cases but the testing data as well, to support the automatic execution of the same applications for hundreds of times with different input parameter values.

Based on the design and implementation of MBT in PHANTOM, EGM is under the preparation of an innovation patent.

4.6.3 Early/Full Prototypes functionality

Early-First Year Prototype

As summarized in Table 4-6, the MBT workflow in Figure 4-4 has been implemented covering all steps, and the following functionalities of MBT in PHANTOM have been developed and demonstrated during the M18 EC Review.

- **Model Validation** for telecom, HPC and surveillance use case applications.
- **Functional Testing** for telecom, HPC and surveillance use case applications.
- **Non-functional testing** for HPC use case applications.

Full Prototype and Next Steps

The following points are currently under investigation and development, and will be delivered in D3.2 in 2018.

- **Performance Testing.** Performance testing, as an aspect of non-functional testing, will focus on the non-functional requirements of use case applications, and interacts with Monitoring framework to execute applications/components and get performance information to evaluate if the non-functional requirements of use case applications are correctly implemented and satisfied.
- **Security Testing.** Security testing, as an aspect of non-functional testing, will focus on testing the implementation of access control mechanism over PHANTOM. MBT simulates the application/component behaviours and sends access requests to PHANTOM security server, and then testing results are generated by comparing the access authorization replies to the predefined access control policies.
- **MBT-MOM interaction.** MBT will provide a specific strategy to provide mapping references with PHANTOM MOM. At the early stage, the MOM requires an initial mapping reference from MBT regarding the performance information between computing tasks and HW. This initial reference will be provided by the ongoing development of the specific MBT-MOM strategy introduced in D2.1.
- **Early Testing in PHANTOM.** In addition to model validation, we are designing another early testing strategy to enable developers to test very early in the life cycle system their application performance over PHANTOM system. The general idea is to take advantage of the component-based characteristics of PHANTOM applications, and estimate the performance of a new application by use of previous MBT results and the way how components are composed together.
- **Updated Functional Testing.** Along with the later use case development and integration with PHANTOM, new functions may come and interfaces may

change. MBT will also take into account the updates and integration of use cases and provide an updated functional testing.

5. CONCLUSION

In this document is reported the current status of the initial development of the tools and technologies that will support the activities of three modules of PHANTOM. Those components are the Parallelization Toolset, the Programming API and the Model Based Testing. For each module are identified:

- the requirements from the use cases that each module must fulfil;
- the design aspects and decisions taken during specification;
- the technical details and technologies used in the implementation of those tools;
- the results of the first tests;
- how the developed tools will communicate with the other PHANTOM modules; and
- the next steps planned for the development the three modules.

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 parallelization technology (e.g. MPI, OpenMP, etc.) that allows a more efficient implementation of the deployment plan and injection of the API/annotations on 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. Future developments of these will focus on the heterogeneity support, namely on the support of GPU and FPGA devices.

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 different components without data exchange. Moreover, an API for specifying CPU-GPU communication was also described. The next steps of the Programming Interface will be in the integration of the CPU-GPU communication API with the OpenCL library and the development of FPGA related support.

Model Based Testing 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, functional testing and non-functional testing for all three use case applications; current MBT results have been provided to developers and presented. The MBT integration in PHANTOM was also defined according to the integration objectives. In the future, MBT will focus on performance testing with monitoring framework, security testing for access control, MBT-MOM interaction, early performance testing strategy and updated functional testing.

6. REFERENCES

- [1]. Terence Parr. The definitive ANTLR 4 reference. Pragmatic Bookshelf, 2013.
- [2]. Chris Lattner. clang: a C language family frontend for LLVM. <http://clang.llvm.org/>, 2016.
- [3]. Developer Express Inc. CodeRush for Visual Studio. <https://www.devexpress.com/products/coderush/>, 2016.
- [4]. Jean-Noël Rouvignac. The AutoRefactor project. <http://autorefactor.org/>, 2014.
- [5]. Xilinx Zynq-7000 All Programmable SoC Product Brief - <https://www.xilinx.com/support/documentation/product-briefs/zynq-7000-product-brief.pdf>
- [6]. SDAccel Development Environment - <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>
- [7]. PHANTOM Linux Software Distribution - <https://github.com/PHANTOM-Platform/PHANTOM-FPGA-Linux>
- [8]. Association of COINS Compiler Infrastructure. COINS Compiler Infrastructure. <http://coinscompiler.osdn.jp/international/>, 2016.
- [9]. Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.
- [10]. U Banerjee. Dependence analysis, 1997
- [11]. S Horwitz, P Pfeiffer, T Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 1989
- [12]. Z Li, PC Yew, CQ Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*. Volume: 1, Issue: 1, 1990
- [13]. Randy Allen and Ken Kennedy, *Optimizing compilers for modern architectures*, Morgan Kaufman Publishers, San Francisco, CA, 2002.
- [14]. Michael Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989.
- [15]. Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997
- [16]. Eclipse Formal Modelling Project, <https://projects.eclipse.org/proposals/eclipse-formal-modeling-project>
- [17]. TTCN-3, <http://www.ttcn-3.org/>
- [18]. Faivre, A., Gaston, C., Gall, P.L.: Symbolic Model Based Testing for Component Oriented Systems. In: *Testing of Software and Communicating Systems*. pp. 90–106. Springer, Berlin, Heidelberg (2007).
- [19]. Eclipse Titan, <https://projects.eclipse.org/projects/tools.titan>.