PHANTOM

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

Project Number 688146

D1.4 – Final design for Cross-layer Programming, Security and Runtime monitoring

Version 1.0
2 April 2019
Final

Public Distribution

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

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

PROJECT PARTNER CONTACT INFORMATION

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

DOCUMENT CONTROL

Version	Status	Date
0.1	First internal draft	09/11/2018
0.3	Further updates and additional content	03/12/2018
0.6	Partner contributions and updates	21/01/2019
0.8	Further contributions and editing of all sections	25/02/2019
0.9	Internal Review	28/03/2019
1.0	Final Release	02/04/2019

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

EXECUTIVE SUMMARY

This document describes the final design of the PHANTOM technologies for the development and execution of applications in a highly heterogeneous and distributed computing environment. The design covers specifications of the core PHANTOM technologies such as a cross-layer programming environment, automatic deployment and execution, Model-Based Testing, integrated security, and automatic runtime monitoring. The specifications that were elaborated by deliverable D1.2 and extended by deliverable D1.3 have been finalized according to the results of validation of the tools (such as the system software for multi-dimensional optimization of D2.2, programmer and productivity oriented tools of D3.2, deployment tools, monitoring framework and FPGA Linux environment of D4.4, model-based testing environment of D3.2). The validation was performed by the consortium partners – providers of the pilot application use cases (Intecs, GMV, and USTUTT-HLRS).

Most notably, the finalized specification includes details of:

- **The PHANTOM Programming Model** – a methodology of component-based application software creation, facilitated by an application and component specification and a deployment configuration. The model was defined in the first phase of the project and has provided the underpinnings of all development from that point. It has been finalized in this document with additional guidelines and API extensions that enhance the user experience and facilitate the final deployment of the application.

- **The PHANTOM Parallelization Toolkit** – an automatic parallelisation tool that analyses component source code and transforms it according to parallelization directives for each of the targeted hardware platforms abilities. The tool selects the best-fitting parallelization technology (i.e. OpenMP for multi-core CPUs, CUDA for GPUs, HLS for FPGAs) and implements it according to the instructions in the Programming Model. The Code Analysis and Technique Selection technologies have been extended to coordinate with the other PHANTOM tools using the updated versions of the Execution and Application Managers.

- **The PHANTOM Multi-Objective Mapper** – a technology that elaborates deployment configurations for application components developed with the PHANTOM Programming Model. Maps over heterogeneously distributed infrastructure in a way that maximises the functional and non-functional requirements of applications. MOM has been aligned with the specification of the Programming Model and its functionality has been generally improved.

- **The PHANTOM Monitoring Framework** – a middleware layer on top of the hardware platform that allows obtaining, storing, and analysis of a wide set of metrics that characterise non-functional behaviour (timing, energy, etc.) of the application and platform. The Monitoring Framework design was enhanced by adding the support for monitoring Nvidia GPUs and Xilinx FPGAs and providing flexible customization options to target any potential hardware, following a plug-in based architecture design.

- **The PHANTOM Deployment Manager** – a technology that performs the final transformation of the application's source code by injecting the communication functions and other low-level functionality in order to enable the execution of the application components on the assigned resources of the heterogeneous distributed system. The initial algorithms of the Deployment Manager have been improved to ensure a better application performance during both compilation and execution.

- **The PHANTOM Security Framework** – an approach to a selected set of security concerns that are brought about by, or emphasized by, the PHANTOM ambition to use heterogeneous processing elements in a distributed system with heterogeneous operating environments. The focal points of the security approach are execution integrity and flexible system access control policy specification. The design of the Security Framework has been refined to better tackle isolation and access control issues.

- **The PHANTOM Model Based Testing** – model based testing for both PHANTOM applications and application components with a focus on functional and non-functional properties. MBT consists of two parts: early validation to test applications in parallel with development, and test execution after applications are complete. In addition to model validation as an early validation method for functional properties, the design is extended by a performance estimation for early non-functional validation. During test execution, security testing is also added as another non-functional testing objective.

- **The PHANTOM Repository –** an integration data layer that can store any files that are required for the application, such as source files, input files, the application and system description files. Additionally, the Repository allows the PHANTOM tools to function on different environments by enabling the storage of files for the internal interactions between the tools (e.g. store alternate versions of the original components generated by the PHANTOM Parallelisation Toolset, the produced binaries that are executed, the deployment mappings etc.). The Repository provides a set of RESTful interfaces which allow both the users and the platform tools to abstract from the distributed and heterogeneous nature of the targeted hardware and serves a one-step communication hub for all inter-platform communications.

- **The PHANTOM Execution, Application and Resource Managers** – the dedicated PHANTOM components that achieve the integration within the platform. They are the latest additions to the PHANTOM design that serve all communication purposes between the user and the PHANTOM tools, as well as within the latter by endorsing service-oriented communication standards (such as REST) to coordinate automatic execution of the tools that are required for each development flow. This removes burden from the developer and increases ease of use.

This deliverable constitutes the final design of the PHANTOM framework as modified by the relevant feedback obtained during integration with the Use Cases. The further content of the deliverable is organized as follows. Section 1 presents the motivation that guided the whole implementation of the design, as well as an overview of the whole toolflow along with its interaction with the user. Section 2 displays a summary of the final design of each individual tool, while details are given regarding the final enhancements done, highlighting major improvements. In Section 3, innovations that extended the current status of the technologies are presented along with way that they contribute to the fulfilment of the project's ambitions.

1. INTRODUCTION

1.1 MOTIVATION

The purpose of the PHANTOM Project toolkit is to help users parallelize and deploy their applications efficiently in heterogeneous hardware, which is a difficult task for many users due to the necessary development time and specialist knowledge.

To fulfill the project's objectives, the tools parallelize application code, and perform the allocation of tasks to the best-suited available hardware in order to improve application efficiency. Efficiency goals are defined and prioritized by the end user, in terms of energy, execution time, etc.

Efficient selection of hardware components benefit work areas ranging from HPC to embedded systems. As an example, the current efforts of device manufacturers to achieve energy efficiency are based on the use of heterogeneous hardware. This is achieved by adding additional hardware components designed for specific tasks, such as DPSs, video transcoders, arithmetic units, vector processing units etc. The improvement that a certain hardware provides is dependent on the application. That is why assigning tasks to the most convenient hardware becomes a challenge. It is even more difficult to distribute applications when hardware availability is limited.

PHANTOM provides an automated solution to this challenge. To achieve this, the tools perform an automatic analysis of the instrumented user's code, using monitoring metrics collected from the hardware where the applications are executed, as well as of the instrumented security levels and eventually Verification & Validation (V&V) via model-based testing.

Security instrumentation is required when applications are running in a distributed system environment, especially when the system is shared with other applications and/or users. V&V is performed in two stages as early validation and test execution to ensure functional correctness and non-functional optimization.

As a summary, Figure 1 outlines a case where the PHANTOM platform has deployed two different applications on two middleware layers. This example shows the flexibility of the PHANTOM platform for using different types of hardware. In particular, the hardware of the system on the left is composed of a homogeneous system of processing nodes, while the middleware layer on the right is composed of a highly heterogeneous system. The Figure shows how in both cases the hardware and software elements interact in the same way with the management, monitoring, and security mechanisms provided by the PHANTOM platform.

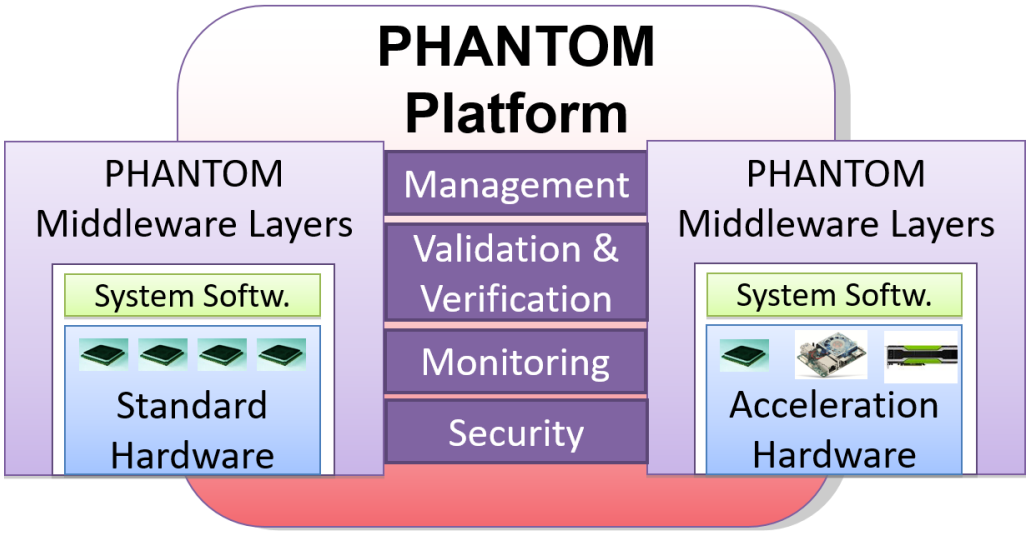


Figure 1: PHANTOM strategy of addressing heterogeneous computer continuum

In order to achieve the goal of simplifying the task of programming heterogeneous platforms, the PHANTOM approach defines a component-based *Programming Model* inspired by the concept of microservices, with functional and non-functional requirements (like timing, communication costs, or security model) as a first-class part of the model.

Figure 2 shows the sequence of stages followed from the developer's inputs to the execution of the developer application.

A developer working with PHANTOM follows these steps:

1. The designs for the intended implementations of PHANTOM applications are first tested by MBT early validation activities from both a functional perspective (based on *model validation*) and non-functional perspective (based on *performance estimation*) without the need to run real applications.

2. The Parallelization Toolset (PT) performs a *Code Analysis* and parallelization *Technique Selection* which invokes source to source transformation for parallelization. Techniques are selected based on the capabilities of the target platform.

3. MBT provides functional and non-functional test cases and several MBT components to support further test execution.

4. The Multi-Objective Mapper (MOM) tool proposes an *optimised deployment plan* for the components of the application depending on the results obtained from MBT, any optional requirement from MBT (e.g., placement restrictions for some components), the user's inputs, and the available hardware resources. Additionally, MOM can consider any existing monitoring data if the application has been previously monitored.

5. Code for execution is prepared by the Deployment Manager (DM), which performs a *code refactorization* defining the most efficient communication between the application's components for the defined deployment plan. It is followed by the compilation of the components, and the upload of the resulting compiled files at the respectively selected hardware resources. Optionally, MBT can register in the Execution Manager the requirement of running certain app's component over certain hardware. After that, the MBT test cases (during testing scenario) and application are executed, and the *application outputs*, the *monitoring metrics* and the *security logs* are collected.

At this point, the developer can iterate their design very easily. Now that monitoring data has been collected, deployment plans produced by the MOM can be more accurate. For each new deployment, the PT selects the corresponding parallelized versions of the application components, and the Deployment Manager ensures that suitable communications code is used so that components coordinate with a minimal amount of overhead.

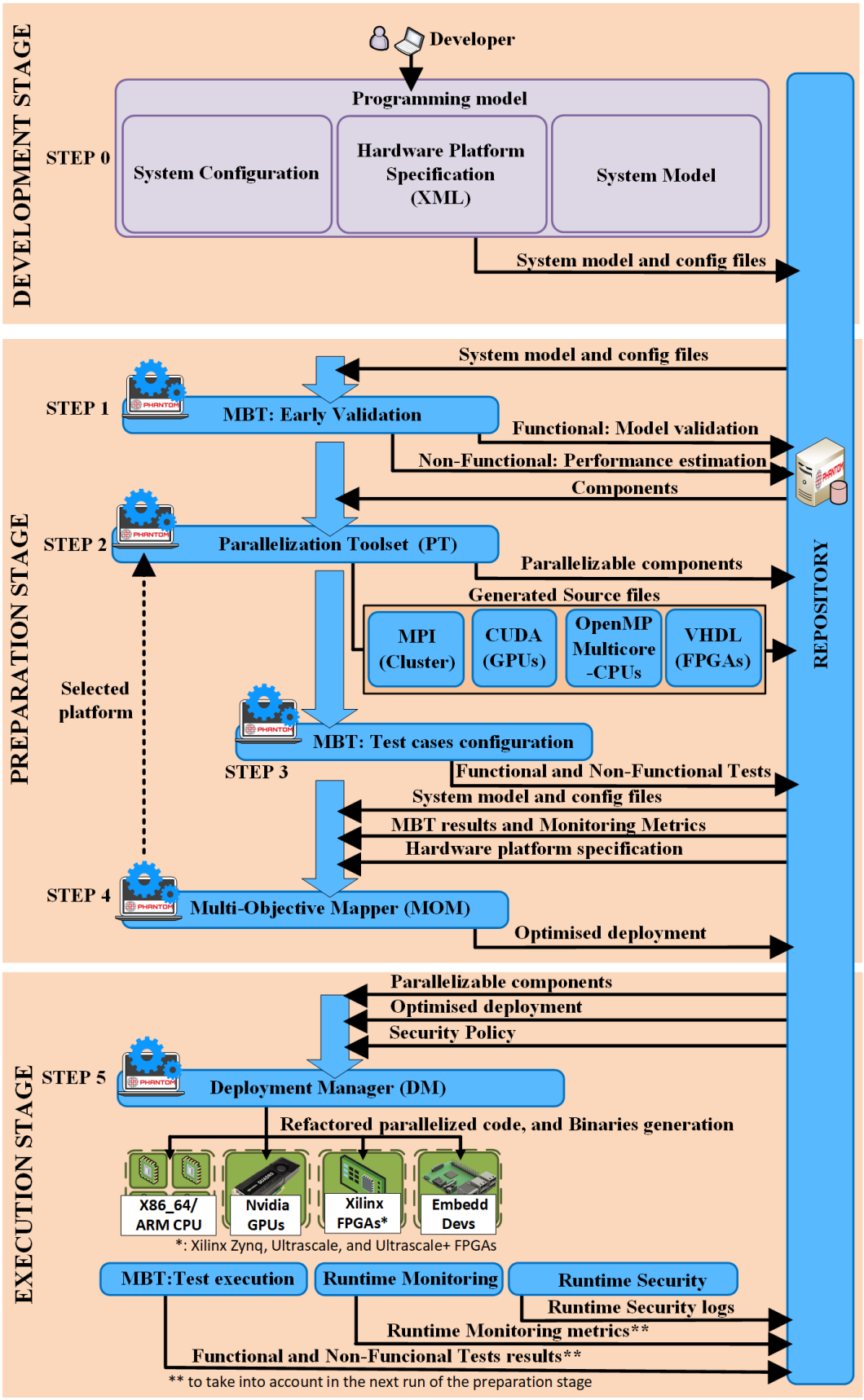


Figure 2: Use of the PHANTOM toolset on five steps.

1.1.1 Final Toolflow Design

Deliverable D1.3 described the *Enhanced design for Cross-layer Programming, Security, and Runtime monitoring*. This is extended and finalized in this section, to the final design of the components of the PHANTOM platform.

Figure 3 shows a compact representation of the flow of the PHANTOM platform components. The top half of the figure shows the *Static* part, which starts with the user's inputs, labeled as the *Programming Model*. These inputs are analyzed and the application components' code is restructured, appropriately annotated, and later parallelized by the tools. This is followed by the development of an optimized deployment plan depending on the user's requirements.

The lower part of the figure shows the components' interaction during the runtime of the transformed user applications.

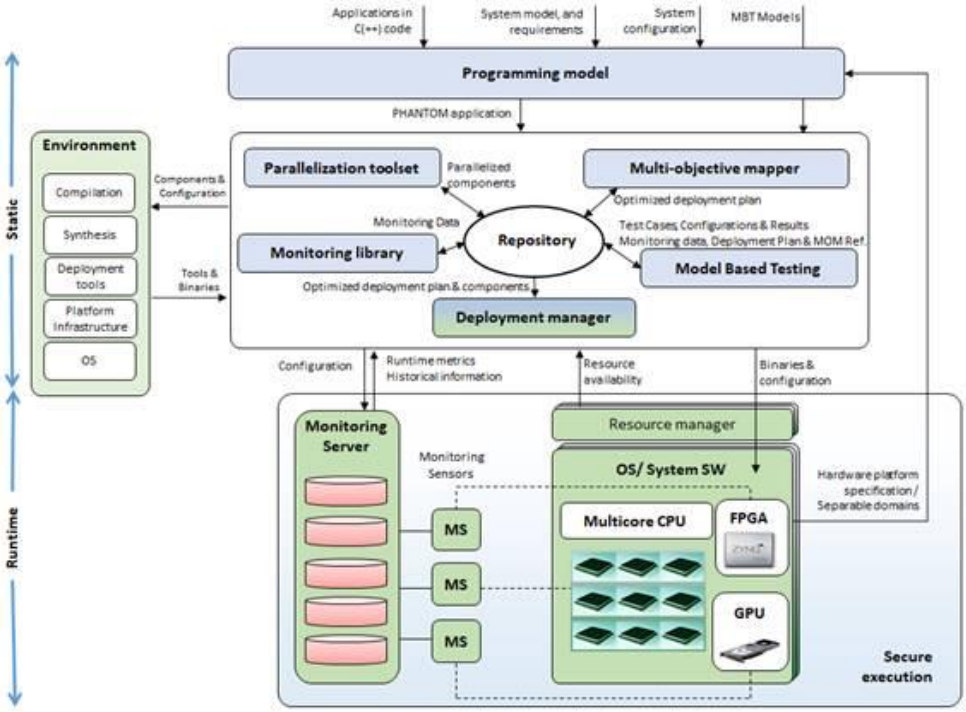


Figure 3: Details of the PHANTOM platform components

Below is a brief description of the functionality of each of the tools. Improvements of the PHANTOM technologies as motivated by feedback obtained from the tools' integration are also mentioned.

Programming Model

The programming model leverages a functionally-oriented component-based methodology to the application design. The developer structures their application as an interconnected network of communicating software components, each of which follows specific guidelines. Along with the application source code, the user provides an initial deployment of their application, which defines where the components of the application

may be located in the target hardware. This may be underspecified, allowing the platform to optimise the deployment according to monitoring and testing results. The developer also defines non-functional requirements of the components, in terms of response time, power usage, security, or many other application-specific metrics.

Parallelisation Toolset

This tool analyses the component's source code and transforms it according to parallelization directives for each of the targeted hardware platform's abilities. The current version of the Parallelization Toolset design concentrates on two main functionalities: Component Parallelization and Parallelization Technique Selection. Parallelization can be achieved through injection of parallelization technology (i.e. OpenMP, CUDA) annotations and generation of IP Cores, aiming to the specific platform that the components are executed on. The code analysis part of the tool runs some dependence tests on the component code, produces parallelized versions and provides information to the MOM about its parallelization capabilities. If the Code Analysis detects that there are components which target a FPGA board, the IP Core Generator is automatically launched to proceed with the component transformation and IP core generation. After the production of the deployment plan by the MOM, the second part of the tool, Technique Selection, uploads the corresponding version of each component to the Repository depending on MOM's decision for the Deployment Manager to proceed with the deployment of the application.

Repository

The PHANTOM Repository server interfaces between the different PHANTOM tools, storing files and the metadata related to them. The metadata is a flexible data structure in JSON-format for each uploaded file, and provides a flexible support for the interaction and integration of the users and PHANTOM tools, and the later among them. The metadata is stored in an ElasticSearch database that allows PHANTOM tools to perform fast search for information of files without need of download files or their metadata.

Multi-Objective Mapper

The Multi-Objective Mapper (MOM) has the task of defining an optimal deployment plan for a PHANTOM application. This implies that MOM needs to take decisions on the mapping of the parallelizable components and on the shared data communications throughout the target hardware architectures (CPU, GPU, FPGA). Decisions are based on:

i. The component network specification of the PHANTOM application

ii. Information regarding the ability of components to be parallelized (from the Parallelisation Toolset)

iii. The target hardware platform specification

iv. User defined functional and non-functional requirements

v. Estimations on the non-functional requirements (from the Model Based Testing)

vi. Performance data from previous application deployments on the underlying hardware platform (from the Monitoring Framework)

vii. User defined estimations on the non-functional requirements

The MOM creates an optimised deployment plan to specify the deployment details of unmapped components to parts of the hardware. While considering the user defined requirements, the generated deployment plan specifies the following variables: host processor of each component, memory location of shared data, directives to PT to select the most effective parallelization technique, mapping of data transfers to memory locations and platform-specific settings.

The MOM can change the values of these variables to attempt to better meet the optimisation goals provided by the developer in the system configuration. In case of conflict, MOM will focus on the highest-priority goal. In the PHANTOM toolchain, MOM will be based on estimations and metrics (on the above variables and the non-functional requirements) that can be determined by the use of Model Based Testing and the Monitoring Framework respectively.

Monitoring Framework

The PHANTOM monitoring framework supports the collection and storage of monitored metrics based on hardware availabilities and platform configuration. The target platform of the monitoring client includes CPUs, GPUs, ACME power measurement kit and FPGA-based platform.

Deployment Manager

The Deployment Manager is responsible for the final code generation / refactoring stage which adds the communication and consistency code that is required by a given deployment as well as building different parts of the project on the selected hardware platforms specified by the Optimized Deployment Plan. Specifically, the Deployment Manager generates the necessary files for the communication among components by using the corresponding techniques (MPI and POSIX functions). Furthermore, the DM generates the scripts that become available to the user for the compilation, linking, and deployment of the binaries on the hardware infrastructure (CPU, GPU, FPGA).

Security Framework

Security Tool 'ngac'

'ngac' is a command-line tool that tests security policies during their development. The tool is able to read and interpret files containing access control policies expressed in a declarative version of the NGAC policy framework. The policies are read from ordinary text files that may be edited with an editor of choice. The tool provides the ability to load policies, combine policies, query resultant policies, and view aspects of policy calculations.

Security Server 'ngac-server'

The 'ngac-server' security policy server provides the Policy Decision Point and Policy Information Point of the NGAC functional architecture within the access control runtime framework. This is a lightweight and very portable implementation of the NGAC standard. It has a number of command-line startup options, and provides both a Policy Query Interface to be called by Policy Enforcement Points, and a Policy Administration Interface to be called by privileged processes in the execution framework to cause policies to be loaded/unloaded, combined, and dynamically modified.

Component Network Execution Integrity

The PHANTOM framework ensures the execution integrity of computational processes deployed on heterogeneous PHANTOM processors (OS-CPU, OS-CPU-GPU, OS-CPU-FPGA). Host OS process and resource management are employed to provide process/memory isolation, controlled interprocess communication, controlled memory sharing, controlled temporal isolation of GPU computations, and controlled communication with FPGA resource sharing infrastructure.

Model-Based Testing

Model-based Testing (MBT) is used to carry out early validation and black box testing for use case applications on the PHANTOM platform focusing global functional and non-functional properties. It reads design specifications, including functional and non-functional requirements, system behaviour descriptions and PHANTOM network component descriptions

MBT in PHANTOM is designed as four activities - model validation, performance estimation, functional testing and non-functional testing - to ensure functionality and performance of applications. MBT has two phases - *early validation* and *test execution*. Early validation is conducted in parallel with application development without executing concrete applications, during which model validation simulates the models created following design specifications and performance estimation evaluates the performance information of an application by analysing composition patterns of components. Test execution executes the functional and non-functional test cases generated from MBT models and provides test verdicts as feedback.

During early validation, the outputs are functional validation results and performance estimation results. In test execution phase, the outputs are functional and non-functional testing verdicts indicating if test cases pass or fail.

1.2 USER INTERACTION

The PHANTOM platform represents a wide range of technologies, both online and offline. In order to deliver the goals of the platform, it is necessary for these tools to interact, both with each other and with the user.

Most of the inter-tool interactions are handled by the PHANTOM Repository described in section 2.2. Tools have clearly defined inputs for their correct operation; these are detailed in the documentation of each tool. This facilitates the interoperability design

through the use of an intermediate storage system. For all repository interactions, a RESTful specification is elaborated, and each of the tools is responsible for uploading their results and necessary content for the other tools to the shared repository.

The following sections describe the interfaces and interaction of the PHATOM tools between them, and also their interaction with the user.

1.2.1 Monitoring Framework

User/developer:
The user only needs to add the instrumentation to their code, and user defined metrics where the user may consider.

Administrative User:
The administrative users need to install a single instance of the MF-Server in a dedicated centralized node for the collection of data. Also, has to install the MF-Client on each device to be used for running the PHANTOM applications, setup a default monitoring configuration on the Resource Manager, and start the MF-Client on such devices.

For each newly-installed device, it is recommended to perform a configuration by means of the PHANTOM Resource Manager. The Resource Manager contains settings for each specific type of supported hardware resources (e.g. x86 CPU, ARM-based CPUs, etc.), which are used to tailor the algorithms of monitoring thus improving its quality.

1.2.2 Repository

User/developer:
The user needs a *user*_id and a password registered by the admin at the Repository, and may also a policy domain name where the user is included *IF* the Repository instance is running integrated with the security manager (it is optional). Users also need the network address of the Repository.

Users need to provide an authentication token (encrypted text string) when they wish to access to the Repository. Users can obtain, providing their user_id and password, a new token at any time from the Repository.

Administrative User:
The administrative users need to install a single instance of the Repository in a dedicated centralized node. Also, the admin has to register, for each user, an *user*_id and the respective password. The admin is also in charge to notify to the users the network address of the Repository, and also the user policy domain names, when the Repository instance is running integrated with the security manager.

1.2.3 Resource, Application, and Execution Managers

User/developer:
Users can query the Managers about the current status of the Devices, Applications, and statistics of the previous executions. Users can also subscribe on updates to the Web-Socket service of these managers.

For such access, each user need a user_id and a password registered by the admin (we consider mandatory be same as the registered for the Repository), and may also a policy domain name where the user is included *IF* the managers' instance are running integrated with the security manager (it is optional). Users also need the network address of the managers.

The access authentication consists on the users provide an authentication token (encrypted text string) when they wish to access to the managers. Users can obtain, providing their user_id and password, a new token at any time from the each one of the managers.

Administrative User:
The administrative users need to install a single instance of each of the Managers in a dedicated centralized node (recommended to be on the same one due to their lightweight load, but they can be install on independent nodes). Also, the admin has to register, for each user, a user_id and the respective password.

Normal users develop applications and use the PHANTOM framework to deploy them effectively according to variable objectives. Some users have a special role, such as those who establish the policies concerning how the infrastructure works, how it provides protections to user applications, and how it places reasonable limits on what a normal user can do. Administrative users install the components of the PHANTOM framework and establish appropriate ownership of executable files, data files, and configuration files to permit the operating system, the PHANTOM tools and managers, and PHANTOM security-specific components (both execution integrity and access control) to work according to their design without intentional or unintentional subversion or interference.

1.2.4 Model Based Testing

User/developer:
For model validation, users and developers create and import the input MBT models to the simulation tool, and the model are simulated and validated with results; for performance estimation, users and developers prepare the component network file of the newly designed application following the PHANTOM specification, and then a command is required so that the input component network file is analysed and a estimation result is presented; for both functional and non-functional testing, users and developers are able to directly execute the test cases (either automatically generated from MBT models or manually defined).

Administrative User:
Besides the above scenarios, administrative users also deploy the MBT systems to support the functionalities of the four activities.

For model simulation and performance estimation, the administrative user installs the model simulation tool and deploys the code of performance estimation; for functional and non-functional testing, the administrative user installs the test execution system so that the whole workflow of test execution can be realized by interacting with other PHANTOM components and the input test cases can be executed. The details for each tool and MBT components are introduced in the next section.

1.2.5 IP Core Generator

User/developer:

The developer adds a pragma to the component source code, identifying which function is intended to run in the FPGA and the Code analysis will recognise the pragma and modify the Component Network accordingly. When the Code Analysis finishes the IP Core Generator runs automatically, reads the Component Network and generates any missing IP Cores for components that can target FPGAs.

In the case of arrays/pointers as function arguments, a set of pragmas identifying which of the arguments are inputs and/or outputs and their respective size, also needs to be added.

Administrative User:

The administrative user needs to install the Vivado HLS and the Xilinx Tools with Zynq support that come with the Vivado Design Suite, in the machine targeted to run the IP Core Generator. The Xilinx Tools need to be properly configured and accessible by the IP Core Generator for proper operation.

1.2.6 User-Scripts

User/developer:

Along with PHANTOM Reference system, a set of scripts was released to support the management and interaction with the system. These scripts help users in the control of the servers deployed on the local environment of the PHANTOM Reference system, as well as provide a mechanism to update all the PHANTOM tools deployed.

These scripts also provide users with a mechanism to upload the user application and all the auxiliary files required for the analysis and compilation of the application. A single configuration file is provided to the where these configurations will be automatically propagated to the PHANTOM tools, being the required tools launched at the appropriated time. A detailed description of the tools and scripts provided can be found in Deliverable D5.2 – Integrated Reference System.

2. FINAL DESIGN OF PHANTOM TECHNOLOGIES

This section describes the final design concerning functionality, performance and usability of the PHANTOM technologies.

2.1 PROGRAMMING MODEL

2.1.1 Overview

In this section, we will summarize the different aspects that define the PHANTOM Programming Model and will present its latest design as it was developed through the enhancements and the needs of the PHANTOM tools during integration. Although the whole picture is described here minimizing the need for accessing more documents to understand the final design of the Programming Model, details about the already mentioned characteristics can be found in the corresponding deliverables.

2.1.2 Design

2.1.2.1 Application and System description Files

As described in previous documents, the architecture of the application created by the user is strictly defined in the Component Network XML file. A PHANTOM application is defined by two different entities, software components and communication objects, as shown in Figure 4.

```
<application-
name="Example_Application"xsi:noNamespaceSchemaLocation="./phantom.xsd">>
<component name="A"  type="asynchronous">
   <implementation id="1" model="any" target-HW="CPU">
      <source lang="c" file="CB.h" path="src\components"/>
   </implementation>
</component>
<component name="B" type="asynchronous" >
   <implementation id="1" model="any" target-HW="CPU">
      <source lang="c" file="CB.h" path="src\components"/>
   </implementation>
</component>
<comm-object name ="B" type="Buffer" object-class="FIFO" size="128" item-size="8">
   <source name="A" port-name="inA1" type="in"/>
   <target name="B" port-name="outB1" type="out"/>
</comm-object>
<comm-object name ="S" type="Signal" object-class="Control">
   <source name="A" port-name="inA2" type="in"/>
   <target name="B" port-name="outB2" type="out"/>
</comm-object>
<comm-object name ="Sh" type="Shared Memory" object-class="Memory" size="1024"
item-size="32">
   <source name="A" port-name="inA3" type="in"/>
   <target name="B" port-name="outB3" type="out"/>
</comm-object>
</application>
```

Figure 4: Component Network example

The Platform Description XML file is used to define the system architecture including all characteristics that are useful for the deployment.

The final design of these files is shown in Figure 5 depicting a CPU-FPGA platform description:

```
<platform>
<device name="CPU-FPGA device" type="CPU-FPGA" reliability="2">
	<processing-node name="UIZynq-unit1" type="CPU-SMP" architecture="SMP">
		<processor name="UIZynq-P1" type="ARM-Cortex">
			<configuration name="core number" value="2"/>
			<configuration name="cpu frequency" value="800" unit="MHz"/>
			<configuration name="bytespercycle" value="1"/>
			<memory name="UIZynq-SM3" type="DDR3" size="1024" size-
unit="MB" access-time="8" access-time-unit="ns/word"/>
		</processor>
	</processing-node>
	<processing-node name="UIZynq-unit2" type="FPGA" brand="Xilinx">
		<fpgalogic name="UIZynq-PL1" type="xc7z045ffg900-2">
			<resource name="UIZynq-LC" type="logiccell"/>
			<resource name="UIZynq-LUT" type="lookuptables"/>
			<resource name="UIZynq-LUTRAM" type="lookuptablesRAM"/>
			<resource name="UIZynq-FF" type="flipflop"/>
			<resource name="UIZynq-BRAM" type="blockRAM"/>
			<resource name="UIZynq-DSP" type="digitalsignalprocessing"/>
			<resource name="UIZynq-BUFG" type="bufferglobal"/>
			<configuration name="UIZynq-maxfrequency" value="200"
units="MHz"/>
			<memory name="UIZynq-SM1" type="DDR3" size="1024" size-
unit="MB" access-time="8" access-time-unit="ns/word"/>
		</fpgalogic>
	</processing-node>

	<local_bus name="UIZynq-AXI" type="AXI4" throughput="15" throughput-time-
unit="Bytes/ns"/> <!--15GB/s-->
	<comm_interface name="UIZynq-EXT" type="Ethernet Network">
		<configuration name="UIZynq-speed" value="100" units="MBit/s"
ip="192.168.1.2" user="external"/>
	</comm_interface>
</device>
</platform>
```

Figure 5: Platform Description Example

Performance, power and security requirements can also be introduced by the user in the Component Network as shown in Figure 6.

```
<requirements name="global_requirements" set-by="USER" target="UC_Surveillance">
   <non-functional   max-value="1000"   measurement-unit="ns"   name="global_WCET"
type="execution-time"/>

   <non-functional max-value="350" measurement-unit="milliwatt" name="global_WCPC"
type="power-consumption"/>
   <non-functional type="security" target-component="Simons"/>
</requirements>
```

Figure 6: Requirement annotations in the Component Network

2.1.2.2 Components

The application is split into different software components that execute in parallel and are launched by the platform at the beginning of the execution. Each component corresponds to a specific source file as defined in the component network, which should have an entry point for the component's execution. Components can be reused multiple times as long as this is explicitly defined in the Component Network.

2.1.2.3 Communication Objects

To enable the communication between the different application components, the PHANTOM Programming Model defines another entity to specifically define the data exchanges or the coordination functionalities that are required by the application. Four types of Communication Objects have been developed, Shared, Queue, Signal, Mutex (described in D1.2), which are implemented by the corresponding interfaces as described D3.2 (Programming Interface).

The protocols have been furtherly enhanced to use specific data types for each communication object, defined by the PHANTOM Programming Interface. In specific, the following types are provided:

- phantom_shared
- phantom_queue
- phantom_signal
- phantom_mutex

The user can **obtain a pointer** to the communication object structure through an initialization method, so that they can call the rest of the API methods using this pointer. The initialization functions are shown below:

```
phantom_shared *phantom_shared_init(char *comm_object_port)
phantom_queue *phantom_queue_init(char *comm_object_port)
phantom_signal *phantom_signal_init(char *comm_object_port)
phantom_mutex *phantom_mutex_init(char *comm_object_port)
```

where *comm_object_port* is the name of the port of the object as defined in the Component Network. The port name is strictly related to the source file and not to the component entity, so components that are linked to the same source file should use the same port name in the component network for the corresponding objects.

The user needs to also declare the corresponding communication object variables in order to use the protocol functions, which are cited below:

Shared
A block of data shared between two or more components. This protocol is mainly used for data exchange.

```
void phantom_synchronize(phantom_shared *item, void *local_data, int dir);
```

Queue
A FIFO queue of fixed size for blocking data exchange between all components. This protocol is used for data exchange and control flow.

```
void *phantom_queue_get(phantom_queue *queue);
bool phantom_queue_put(phantom_queue *queue,void *item);
void *phantom_queue_peek(phantom_queue *queue);
uint32_t phantom_queue_count(phantom_queue *queue);
```

Signal
A protocol used to coordinate the execution of components.

```
int phantom_notify(phantom_signal *signal);
int phantom_wait(phantom_signal *signal);
void phantom_barrier(phantom_signal *signal);
int phantom_notifyall(phantom_signal *signal);
```

Mutex
A protocol used to enforce mutual exclusion, for example to protect the integrity of shared data.

```
int phantom_lock(phantom_mutex *mutex);
int phantom_unlock(phantom_mutex *mutex);
int phantom_trylock(phantom_mutex *mutex);
```

2.1.2.4 Enhanced Programming Model annotations and functions

The PHANTOM Programming Model has been extended with file operations (as described in D1.3). The added functions are mirrors of the POSIX file operations in order to maximise compatibility. The stream objects returned by the functions are compatible with the other I/O functions from the C standard library, fprintf, fscanf, snprintf, sprintf, sscanf.

FILE * phantom_fopen (const char * filename, const char * mode)
int phantom_fclose (FILE * stream)
int phantom_fflush (FILE * stream)
size_t phantom_fwrite (const void * ptr, size_t size, size_t count, FILE * stream)
size_t phantom_fread (void * ptr, size_t size, size_t count, FILE * stream)
int phantom_fgetpos (FILE * stream, fpos_t * pos)
int phantom_fseek (FILE * stream, long int offset, int origin)

Additionally, the following functions are also introduced to fit to the needs of the Use Cases:

int phantom_fileno(FILE *stream)
The *phantom_fileno()* function returns the integer file descriptor associated with the stream pointed to by *stream*.

FILE *phantom_fdopen(int fd, const char *mode)
Associates a stream with the existing file descriptor, *fd.* The *mode* of the stream (one of the values: “r”, “r+”, “w”, “w+”, “a”, “a+”) must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fd*, and the error and end-of-file indicators are cleared. Modes “w” or “w+” do not cause truncation of the file. The file descriptor is not dup’ed, and will be closed when the script created by *phantom_fdopen()* is closed. The result of applying *phantom_fdopen()* to a shared memory object is undefined.

int phantom_get_fd_flags(int fd, int value)
Retrieves the current position in the *stream*.

long int phantom_ftell(FILE *stream)
Returns the current value of the position indicator of the *stream*. For binary streams, this is the number of bytes from the beginning of the file. For text streams, the numerical value can be used to restore the position to the same position later using *phantom_fseek()*.

The programming model has been extended to include assisting annotations for the static analysis of the code to enhance the parallelization results of the Parallelization Toolset. These provide additional information that is too complex to be extracted automatically and therefore can be used by developers to improve deployment of their code, without enforcing guidelines on all component developers.

#pragma function no-side-effects
Function called doesn't have any side effects on global memory (only local). Pointers passed to it though are accessed manipulating data at the corresponding addresses.

#pragma function pure
Function call doesn't have any side effects.

#pragma loop static-loop-bounds
Loop bounds can be statically determined.

#pragma [function | loop] no-pointer-aliasing
Function or loop doesn't include any pointer aliasing when accessing memory addresses.

#pragma [function | loop] no-dynamic-pointers
Function or loop doesn't include any pointers that have dynamically allocated memory space to them.

#pragma [function | loop] static-vectors
Function or loop doesn't include any vectors with dynamically modified size.

The platform will verify these pragmas (when possible) in order to assist the developer, and error if they are violated, but in general they are understood as guarantees from the programmer to the platform.

2.1.2.5 Final enhancements of the Programming Model

To further extend the support of the Programming Model, the Queue protocol has been enhanced to transfer objects of dynamically determined size. To this end, a series of serialization/deserialization functions are offered for the translation of objects into streamed data. For example:

*size_t Serialization::serialize_mat(vector<vector<bool> > *mat, char *out, bool w_size)*

Users are able to provide their own serialization functions according to their needs, but a dedicated library is available by the framework for the user to call at any time. The size of the serialized data is considered in the first 4 bytes of the buffer.

Additionally, a middleware interface is provided for the user to exploit the Monitoring Library:

The *phantom_monitor* type is defined to give access to the monitoring functionalities provided by PHANTOM.

```
phantom_monitor *phantom_monitor_init()
```
Returns a pointer to the monitor that is used to send metrics to the Monitoring Framework.

```
void phantom_mf_start(phantom_monitor *monitor)
```
Registers the start of the component. If not used, the start of the execution will be automatically registered which may not identify exactly the start of the computationally intensive part of the component.

```
void phantom_mf_user_metric(phantom_monitor *monitor, char *metric_name,
int value)
```
Registers a user defined metric that the user may find useful.

```
void phantom_mf_send(phantom_monitor *monitor)
```
Sends the already gathered data by the Monitoring Client to the Monitoring Server and clears the buffer space.

```
void phantom_mf_end(phantom_monitor *monitor)
```
Registers the end of the component. If not used, the end of the execution will be automatically registered which may not identify exactly the end of the computationally intensive part of the component.

2.2 REPOSITORY

The normal operation of the PHANTOM tools requires the exchange of information and files between them. This requirement has to be met even when not all the tools have to run simultaneously, and when the content to be shared may be required multiple times. Therefore, an intermediate system for the storage of such data called the *Repository* is used. The Repository has a clearly defined interface for storing and accessing the different types of content to facilitate its use by different tools.

Among the contents to be stored in the repository, we can highlight the programming model provided by the developer, the test models and their results generated by MBT, different versions of the generated code, deployment plan, and execution results such as security logs and monitoring metrics.

In addition to data storage, the Repository will also provide a pub/sub (publish-subscribe) mechanism for PHANTOM components to subscribe to changes of files on a particular path and get notified when data is updated. In particular, the notification consists on sending the metadata of the modified files. This is useful in scenarios such as PHANTOM users triggering the execution of an application when given input data, and users are notified when the execution is over and output data is ready.

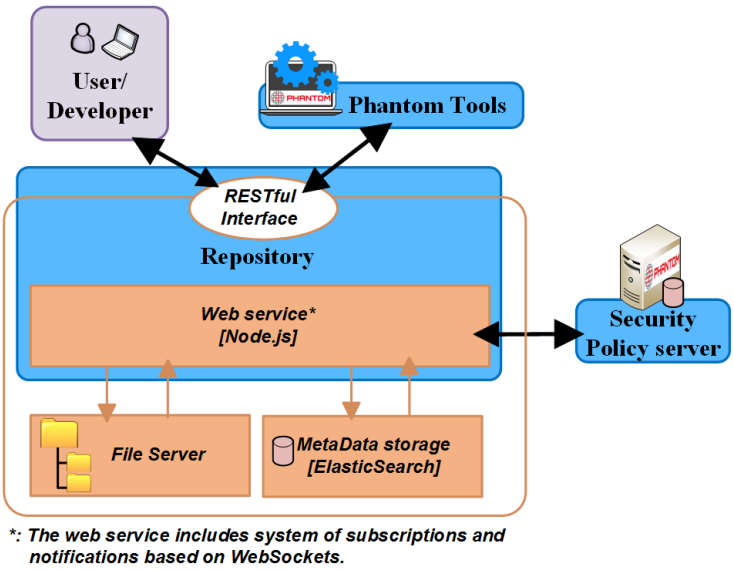


Figure 7: Software frameworks and interfaces of the Repository

2.3 PARALLELISATION TOOLSET

2.3.1 Overview

PT's Code Analysis module takes sequential code and transforms it in order to enable loop and task parallelization. In particular, it identifies all existing loops inside a portion of a components' source code and parallelizes some of the non-complex ones. The tool's Technique Selection module is then used to select the adequate parallelized versions for each analysed source file and uploads it on the Repository. The main task of the tool is to provide important automation of parallelization tasks that would otherwise have to be done manually. In this section, the basic design of the tool is described, whilst details can be found in D3.2.

2.3.2 Design

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

2.3.2.1 Code Analysis

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

- **OpenMP version:** Includes OpenMP annotations that indicate the usage of multiple threads for the component's execution in CPU platforms. The ROSE Compiler's autoPar tool is exploited for its dependence analysis and OpenMP annotation injection properties.

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

The generation of these versions of the components will give the ability to PHANTOM to exploit the available hardware platform and accelerate the application's functionality.

2.3.2.2 IP Core Generator

The IP Core Generator is described in more detail in deliverable D4.4 and here we will only focus on the integration with the rest of the PHANTOM components.

The IP Core Generator runs automatically without any need for user intervention, interacting with the other PHANTOM components via the Repository and App Manager. It subscribes to the PT Code Analysis status in the App Manager and when the Code Analysis is finished, the IP Core Generator downloads the Component Network that was modified by the PT from the Repository and analyses it, searching for components that can target FPGAs. When one is found, the correspondent component files are downloaded from the Repository to proceed with the source code transformation. The transformed source code is then used to generate an IP core. The tool also creates a modified component that interfaces with the IP core and can be used the same way as the normal software component would be used.

Both the generated IP Cores and the modified components are uploaded to the Repository, as well as a new version of the Component Network with the new FPGA implementations inserted, to be used by the rest of the PHANTOM components.

2.3.2.3 Technique Selection

Code Analysis and the IP Core Generator complete the static analysis part of the tool. Technique Selection is included in the iterative part of the PHANTOM toolflow, whereas a Deployment Plan is produced by the MOM determining which version of each component to deploy according to the hardware resources assigned to each component using a straight forward decision mechanism. In particular, for the case that the component runs on a CPU the OpenMP version is used, enabling the possibility of running the component in parallel using multiple threads. If the MOM decides to use a GPU or an FPGA to accelerate the component's execution, the CUDA or FPGA versions are chosen and uploaded on the Repository to be deployed by the Deployment Manager.

2.4 MULTI-OBJECTIVE MAPPER

2.4.1 Overview

The Multi-Objective Mapper (MOM) is responsible for optimising the mapping of components and shared data communications throughout the target architecture, towards user-defined non-functional requirements. MOM considers evolutionary/bio-inspired multi-objective optimization approaches in order to optimize the placement of components against multiple objectives such as power, performance (execution time, power consumption, memory utilization, reliability, data movement, security), considering requirements/policies from application developers' side, as well as the status and capabilities of computing resources.

2.4.2 Design

2.4.2.1 Generic MOM

The design specifications of MOM are provided in details in deliverable D2.2.

The goal of Generic MOM is to optimise the deployment of the user application components to the underlying hardware platforms for user requirements satisfaction and maximum performance in terms of execution time, energy efficiency, security and other use case requirements, as defined in D1.4 and addressed in D2.2. To that end, MOM needs to interact with other components in PHANTOM architecture (as shown in Figure 8 and Figure 9 below), namely the Model-Based Testing, the Parallelization toolset, the Monitoring library and framework, as well as the Repository, from which MOM receives its input and stores its outcome.

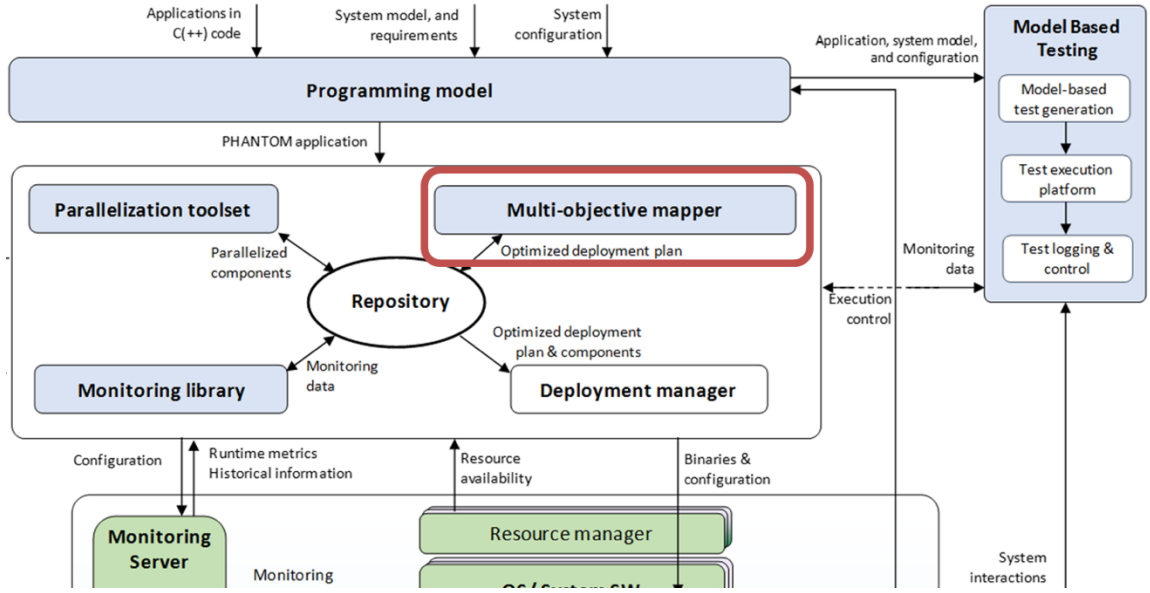


Figure 8: MOM positioning in PHANTOM architecture

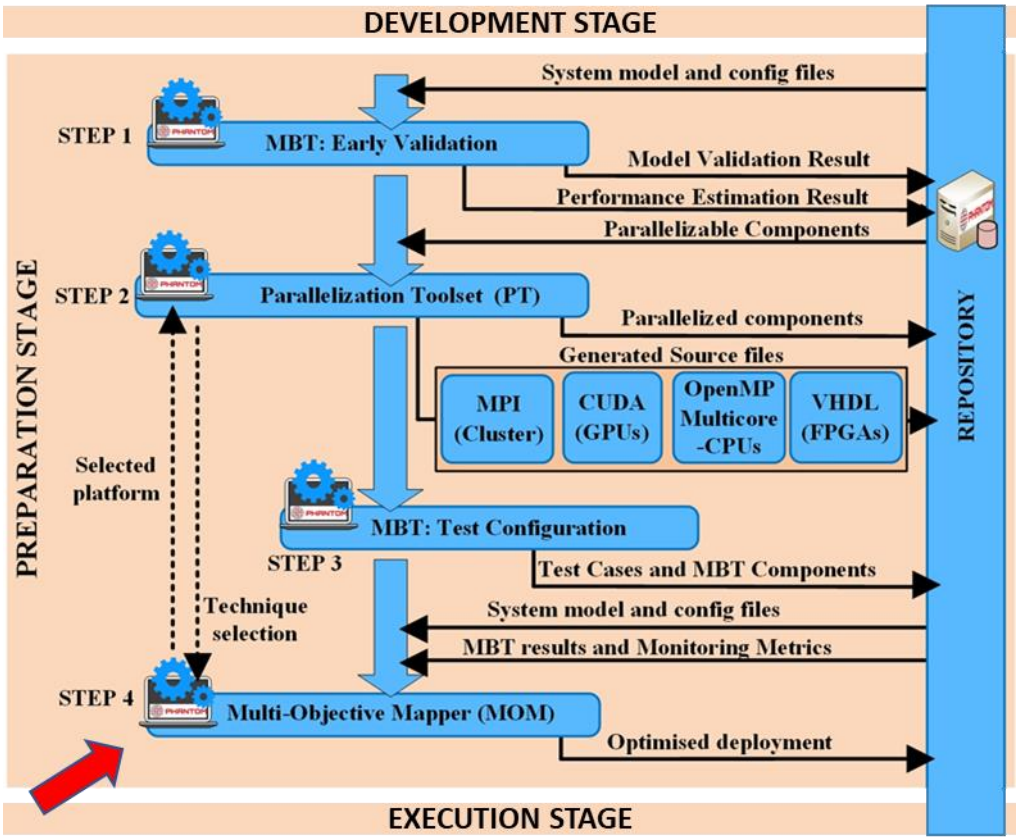


Figure 9: MOM positioning in PHANTOM tool-flow

MOM requires the definition of the hardware infrastructure and the application's description in a PHANTOM-specific format. Specifically, MOM requires as input the System Model (component network specification), the Platform Model (hardware platform specification) and the System Configuration (user defined requirements and, if available, the initial mapping). Furthermore, MOM collects estimations on the non-functional requirements from the Model-Based Testing toolset, while results from previous application deployments on the underlying hardware platform (Performance data) are derived from run-time monitoring through the Monitoring Library. In addition, the Parallelization toolset provides MOM with parallelization directives to further assist MOM in the mapping optimization process. The Multi-Objective Mapper is then performing a number of optimization steps, executing a multi-objective optimization evolutionary/bio-inspired algorithm, which considers all the above input specification and the related requirements. Finally, MOM generates the Optimised deployment plan (mapping decision of application components to hardware platform processing elements and of communication objects to hardware platform physical memories). All interactions that MOM performs with the rest of the PHANTOM components are facilitated and implemented through the use of the Repository. The latter provides secure interactions using a token mechanism to manage the access rights of PHANTOM users.

Implementation details

The optimized mapping is generated from an evolutionary/bio-inspired multi-objective algorithm, inspired by weighted and genetic algorithms, which is optimized towards

PHANTOM requirements. The generic MOM's evolutionary multi-objective algorithm is a custom designed genetic algorithm implemented in Java, using appropriate XML libraries to be able to parse its input and produce its output. The generic MOM receives and processes the component network specification along with the hardware platform specification and also an initial mapping, in order to produce the optimized deployment plan, as described in the following steps:

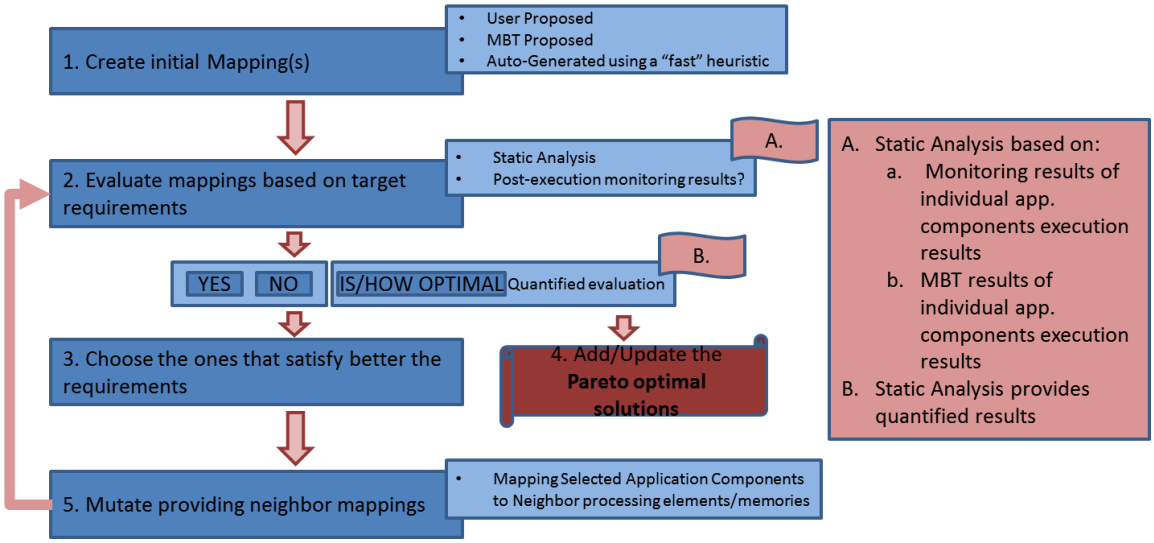


Figure 10: High-level representation of Generic MOM algorithm

Step 1: At the first step, initial mappings are created using a random mapping function. The number of initial mappings is user defined (passed as argument). User may define a custom initial mapping. The satisfaction of all supported requirements is checked upon mapping creation. If a mapping violates the requirements, it is discarded and another one is created to replace it. This applies to the mutated mappings as well. The current supported requirements are user defined and range from total execution time, power consumption, component network connections, underlying hardware characteristics (HW architecture and connections), to memory size (data accommodation capacity).

Step 2: After each mapping is created, an evaluation function is called to evaluate the generated mappings performance based on the given user-defined requirements (time, power, etc.). MOM tool proceeds to a quantified estimation of parameters such as the computation times, the total execution time and the power consumption, based on the metrics (CPU speed, memory size and memory access time) provided by the Platform Model (hardware platform specification) and the Monitoring Framework.

Step 3: The best mapping is selected based on the performance on the given user-defined requirements and is added to the Pareto optimal solutions list.

Step 4: The next step consists of the Mutation of the best mapping providing neighbour mappings, according to the user defined requirements in execution time or power consumption. There are three levels of mutation applied in the selected mapping listed in consecutive order:

Step 4a: The first level of mutation refers to Communication objects remapping based on the memory access time. Communication objects mapped on hardware memories (local or remote) with the highest access time, are remapped to memories with less or the least access time (local memories), in order to achieve less communication time provided by the mutated mappings.

Step 4b: The second level of mutation consists of (1) the identification of the component with the highest execution time that is parallelizable, (2) splitting it into two subcomponents, (3) generation of random mappings based on the modified component network. This operation aims on increasing the component parallelism in the mutated mappings in order to decrease the total computation time.

Step 4c: The third level of mutation targets the power consumption optimisation. It consists of (1) the identification of the component with the highest power consumption, (2) the move in a neighbouring hardware element and (3) the generation of random mappings for the rest of application components. This operation investigates other mapping alternatives for lower power consumption.

Step 5: The mutation loop is executed to populate the Pareto optimal solutions, until the user defined number of iterations is reached or until mutation process cannot provide better mappings for several executions.

Finally, the Multi-Objective Mapper produces the optimized deployment plan that indicates the mapping of application components to corresponding hardware processing elements.

2.4.2.2 Offline MOM

The main implementation details of the Offline MOM are in deliverable D2.2. This section focusses on its interaction with the Generic MOM.

The goal of the Offline MOM is to attempt to apply existing analytical approaches to determine whether timing issues are likely to occur in a designed system. This is done before deployment and testing, in order to assist rapid development and iteration.

Once the Generic MOM has created a deployment, the Offline MOM will analyse it to determine whether any issues with the mapping exist. Timing analysis is not mature enough to be able to perform exact worst-case response time analysis over complex multicore systems with on-chip networks. Therefore, the PHANTOM approach is aimed at a development flow that can integrate the current state-of-the-art, and evolve to use newer techniques as they become available. It is for this reason that the primary function of the Offline MOM is as a necessary condition, rather than a sufficient one. i.e. it can *reject* mappings, but only *verify* mappings when the target platform is simple enough such that there is a suitable exact analytical approach.

The Offline MOM is entirely transparent to the user, and it interacts solely through the Repository. It subscribes to the existing development project, and so is notified every time a new deployment plan is uploaded (either by the MOM or by the user). Whenever a new deployment is added, the Offline MOM downloads it and its associated Component Network and Platform Description. Then a range of analytical tests are applied to the model, looking

for issues with CPU load, network load, and worst case response times. If a system is deemed to have a worst case that is higher than a one of its requirements, then the Deployment is marked as failing in the Repository, along with the reason why. If no tests show problems, then the test is marked as passing. On unpredictable systems with no timing model, such as a shared infrastructure virtual machine, only a limited range of tests can be applied. On smaller and more predictable systems much more accurate tests can be employed. In this way, the Offline MOM compliments the work of the Generic MOM by aiming to spot errors before deployment.

2.5 DEPLOYMENT MANAGER

2.5.1 Overview

The Deployment Manager automates the deployment procedure due to the different modifications that need to occur for the components' execution on complex, heterogeneous environments. The different adjustments and code refinements are executed as a final stage of the application's development following the analysis and results of the Parallelization Toolset and the decisions of the Multi-Objective Mapper. Additionally, a set of scripts corresponding to the aforementioned procedure, is generated implementing the actual deployment of the components on the available hardware. Details about the aforementioned procedure can be found in D4.4.

2.5.2 Design

The Deployment Manager is responsible for the source code modifications and compilation/deployment scripts in order to facilitate deployment on CPU, GPU, and FPGA targets. In this section, these steps are described in detail, while in the following section some lower-level implementation details are included.

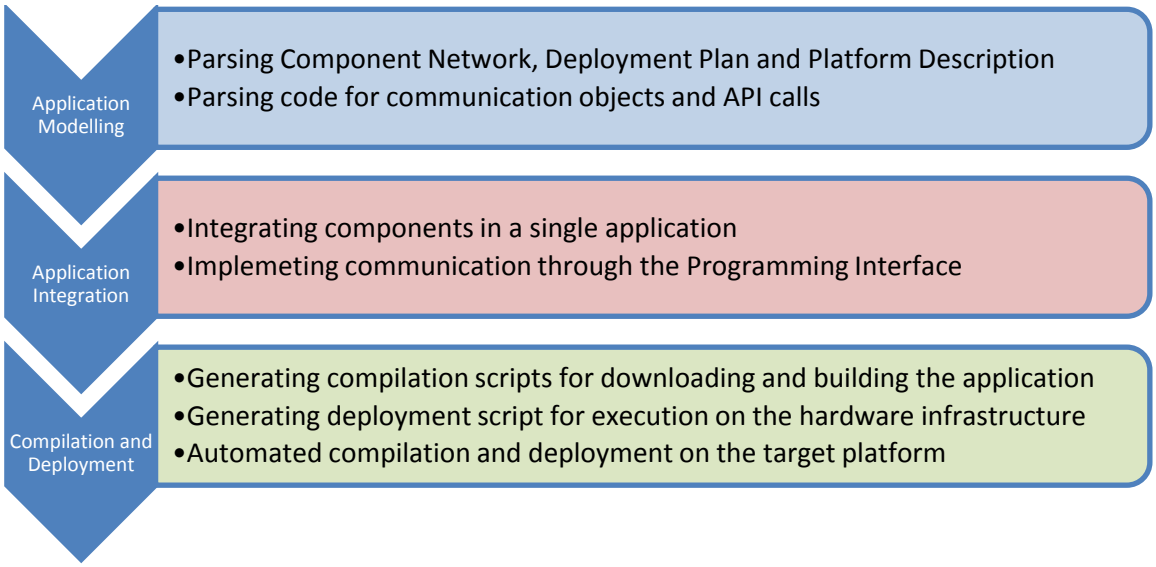


Figure 11: Deployment Manager functionality

As shown in Figure 11, the tool flow of the Deployment Manager can be split into three stages.

First, there is the modelling of the application where a set of Java classes are used to describe the components, the way they communicate with each other, their mapping on the hardware platform and information about the data or signals they need to exchange. The Deployment Manager needs to collect all the necessary information to model the application per component accompanied by the corresponding communication objects. This information is in the Component Network, the Platform Description and the Deployment Plan as well as the source code in the form of pragma annotations. The architecture of the model that is generated follows the structure of Figure 12.

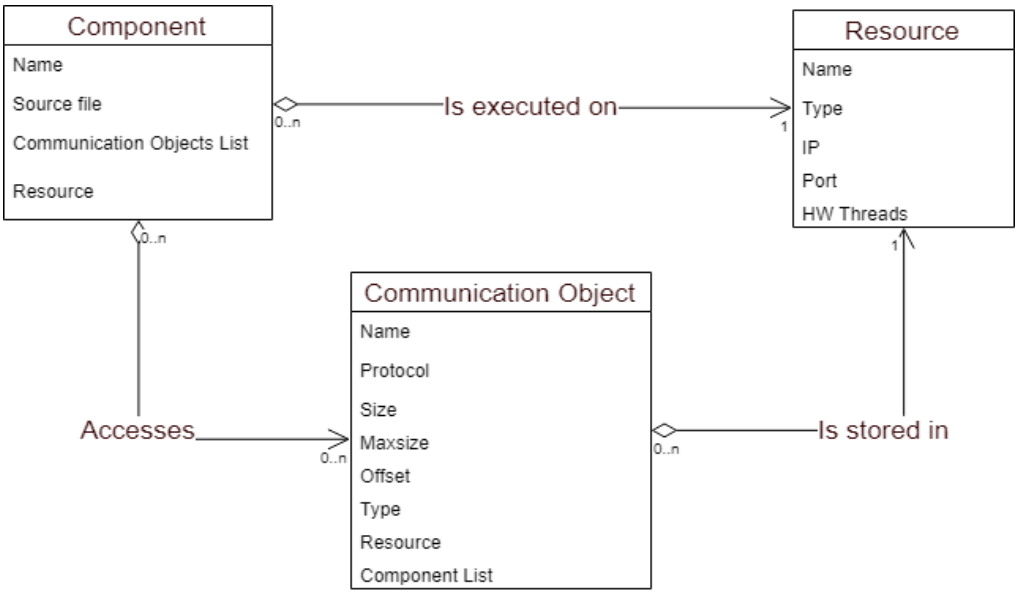


Figure 12: Modelling of the Application by the Deployment Manager

Second, the generation of the necessary files will take place enabling the integration of the components with each other, implementing in this way the communication between the different parts of the application. These files include the invocation of the components declared in the component network and the memory allocation and communication handling needed for the communication between them. Additionally, the Programming Interface source files will also be linked with the application completing the final deployment source code.

Finally, a set of scripts are generated for building and placing the binaries on the corresponding machines. These are executed automatically by the Deployment Manager compiling the source files on the target machines and storing the resulting binaries on the Repository. Specifically for the deployment on the FPGAs the Deployment Manager interacts with the FPGA Linux Infrastructure through the generated scripts to build the IPCores on the FPGA targets as described in the next section.

FPGA Linux Infrastructure

The main implementation details of the FPGA Linux infrastructure are in deliverable D4.4. This section will focus on its interaction with the DM.

The role of the FPGA Linux infrastructure is to support the PHANTOM programming model. In traditional FPGA development, experienced hardware designers are required to

partition an application, create the corresponding hardware design, and write supporting drivers and other software, to make effective use of the FPGA platform. The component-based programming model of PHANTOM eases a lot of this difficulty by compartmentalising communications and processing such that it can be more easily handled by automated tools.

The infrastructure therefore implements the following features:

- An implementation of the standard PHANTOM communications API, which provides abstraction of communications and therefore allows the FPGA board to be part of the distributed PHANTOM application.
- A PHANTOM IP core specification. This specification is a common interconnection standard for PHANTOM-compatible IP cores to use.
- An automated hardware design system that can fabricate systems from a provided list of IP cores that adhere to the IP core specification. Given a list of cores, the system assembles a hardware design, along with associated clock generation, reset logic, and bus and memory arbitration.
- Having generated the hardware design, the supporting Linux kernel, root filesystem, component drivers, software, and device trees are also automatically generated. This builds a complete Linux environment in which software components can interact with hardware components without the designer having to accommodate for this in their code.
- Based on the security requirements of the component, the generated hardware can also be security hardened. This prevents all other components of the FPGA (both hardware and software) from accessing a component's memory space, and from otherwise interacting with its execution. This provides a far higher level of security than other accelerators, such as GPUs.
- Other parts of the PHANTOM toolchain, such as monitoring, are also supported from the FPGA target.

Therefore the DM uses the Linux infrastructure when it has been instructed by the MOM to offload a given set of components to a particular FPGA target. The DM first creates a small configuration file that lists the following:

- The target FPGA board.
- The name of each PHANTOM IP core to include in the design.
- And optionally, an amount of memory to allocate to each IP core. If not provided, memory will simply be shared equally.

The Linux infrastructure then connects to the Repository, downloads each specified IP core, and begins work constructing its outputs. Once complete, the following will have been produced:

- An FPGA bitfile for the target board, which implements a hardware design containing the requested IP cores.
- A bootloader for the target board which initialises the system, programs the bitfile onto the FPGA's reconfigurable logic, and chain loads the Linux kernel.

- A Linux kernel with associated drivers for implementing monitoring and low-level communications.
- A Linux device tree that describes the IP cores in the system.
- A Linux root filesystem with PHANTOM API implementations, and the software parts of all components allocated to the FPGA.

As FPGA development tools can take a long time to execute, this process could take hours on a large FPGA with many IP cores. Also, the FPGA vendor tools must be installed and licensed. Once complete, the outputs can then be copied to an FPGA using whatever method is preferred. Instructions for SD card booting and TFTP booting are included in the software release.

2.6 MONITORING FRAMEWORK

The objective of the PHANTOM Monitoring Framework is to monitor the system load during previous executions of PHANTOM applications (application-level), and the status of the available devices (infrastructure-level), in order to better inform the next execution. The monitoring workflow is divided into two parts as shown in Figure 13. The devices are monitored by the **MF-Client** (Infrastructure-level), and the instrumented applications monitored using the **MF Library** (Application-level). The metrics collected are sent to the **MF-Server**, which provides an interface for query and analysis.

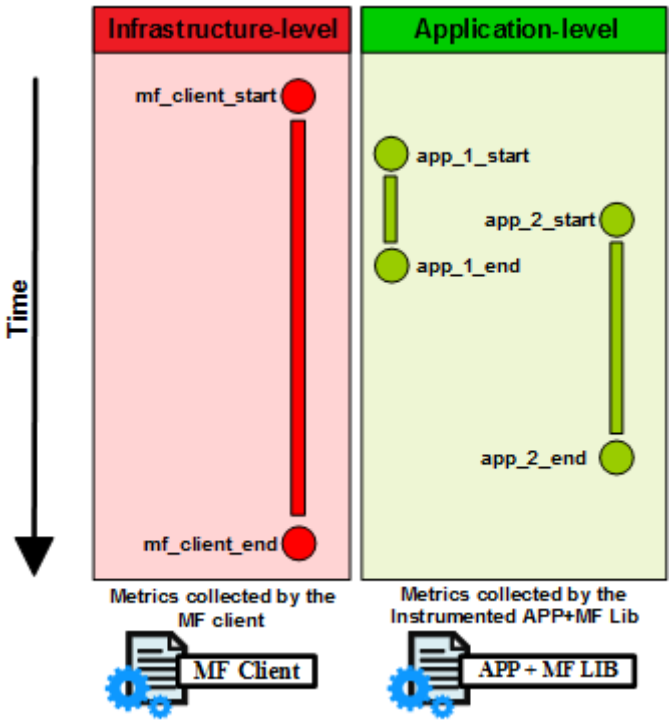


Figure 13: Infrastructure-level monitoring with the Monitoring Client, and Application-level monitoring with apps instrumented with the MF library

Figure 14 shows the client-server architecture of the MF, which is composed of the MF-Server, and the two kinds of monitoring agents: MF-Client and the MF-Library. The figure also shows the information exchanged between those components. The three types

of exchanged information are the *Monitoring Configuration*, the *Monitored Metrics* and the *Current Status*.

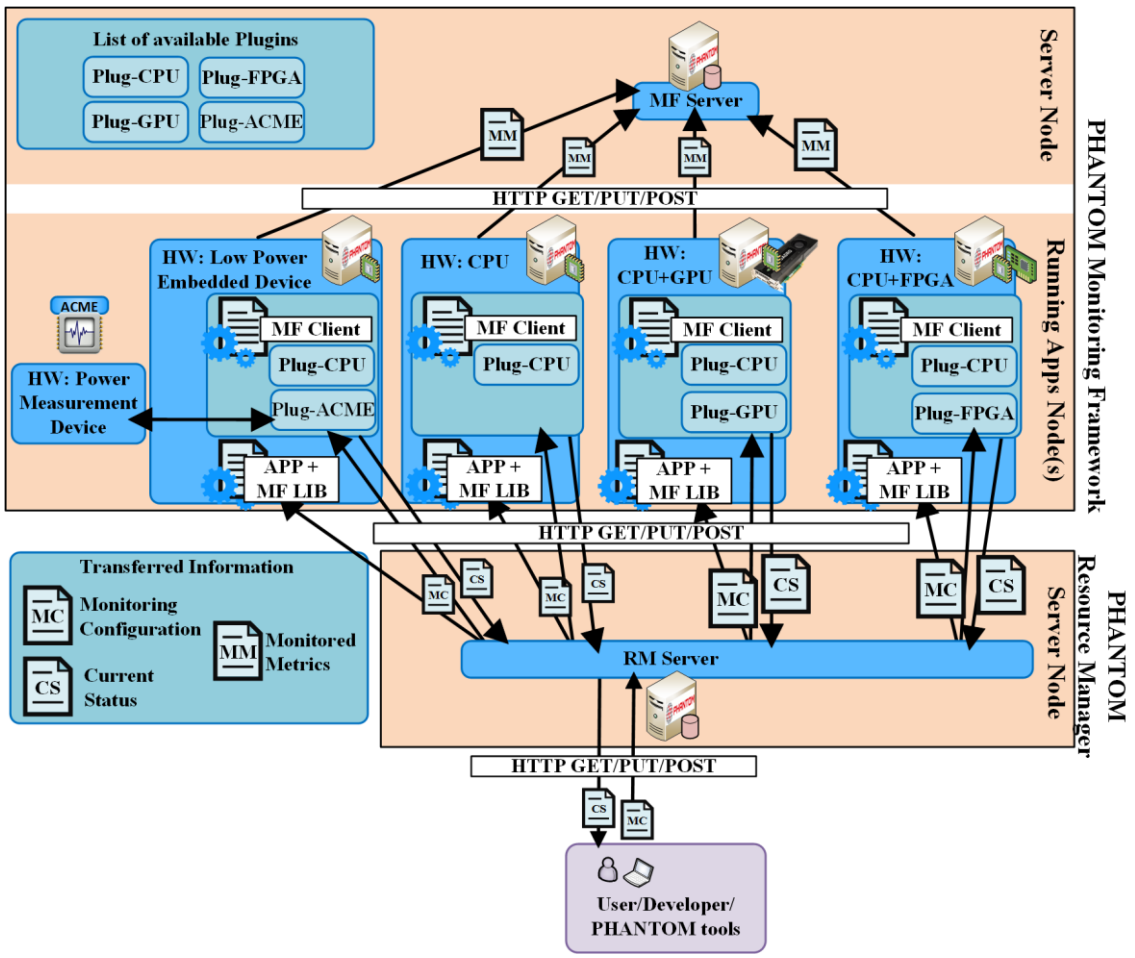


Figure 14: PHANTOM Monitoring Framework Architecture

2.6.1 MF-Server

The MF-server stores the received metrics in a database and offers the functionalities to query and analyse the run-time collected metrics. The customers of the monitoring results are the end-users and the other PHANTOM services. The MF-server is composed of a data storage layer, used to persistently store the monitoring information from the sensors/agents, and a web-service for the data transmission from the agents to the data storage layer.

For the data storage component, ElasticSearch is used. It is a flexible and powerful open-source, real-time search and analytics engine. As a distributed, multi-tenant full-text search engine, it is preferred as supporting RESTful web interface and using schema-free JSON documents.

The Web service is written in JavaScript under the Node.js runtime environment. Node.js has an event-driven architecture and is suitable to be used for data-intensive real-time applications that run across distributed devices. A free and open-source framework for Node.js – Express.js, has been selected for building the RESTful APIs.

2.6.2 MF-Client (monitoring at infrastructure level)

The MF-Client is a lightweight service that is installed on top of the available system software stack (it runs as a separate system process in the background of the OS). The MF-Client task is monitor the state of the device. It is done collecting infrastructure-specific metrics on the whole node/board, regardless of the applications running on it, and forwarding them to the MF-Server.

The MF-Client design provides a flexible implementation of clients by importing plugins. Only required plugins are loaded, allowing for the minimization of resources used on the monitored device. Each one of these plugins is an independent thread that wakes up on its monitoring time interval. The default plugins to be loaded as their default configuration for each computation node are registered at the Resource Manager (RM). The MF-Client supports that the Monitoring Configuration be updated during run-time. The update process consists of the MF-Client load the Monitoring Configuration periodically (the load frequency is also a parameter in such configuration).

The MF-Client collects metrics at the system level by importing and starting the required set of Monitoring Plugins in the device to be monitored. All the available plugins run as independent threads. In order to decrease system utilization, the plugin threads sleep, and periodically wake up on its monitoring time interval for collecting metrics and storing them in an independent local buffer. There is a different time interval for transferring the content of each buffer to the MF-Server/RM, which is done in JSON format.

2.6.3 MF-Library (monitoring at application level)

The PHANTOM Monitoring Library (application-level monitoring and user-defined metrics collection) abstracts users from the metric collection process. The PHANTOM MF-Library provides a user library and several APIs for instrumentation of the users' applications. It allows users to control and adjust the metric collection process by means of the user-level APIs.

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

The instrumented applications with the MF-library load when starting their default monitoring configuration from the RM. This default configuration, defined by the system administrator, allows users to not have to perform additional actions and do not need to know the hardware details of the device where the application will run. However, users can, if they wish, provide a different configuration in their instrumented applications.

2.6.4 Monitoring Configuration

The Monitoring Configuration is stored in the Resource Manager and details the default configuration parameters for the MF-Client and the apps instrumented with the MF-Library. Configuration parameters contains a list of metrics to be acquired, their sampling interval, and the frequency at which they should be uploaded to the Server. For simplicity, many default configurations are provided.

2.6.5 Monitoring Metrics

The metrics that can be acquired cover a wide range of functions that target different aspects of the hardware, such as memory, IO, processor, GPU, and network utilization as well as energy consumption. Metrics are composed of a simple key-value and timestamped.

2.7 SECURITY FRAMEWORK

2.7.1 Overview

The PHANTOM Security Framework has been focused on two areas, identified in D1.2, that reflect distinct and characteristic security concerns in a heterogeneous environment:

- A low-level solution to ensure integrity of component network execution on the heterogeneous PHANTOM platform based on isolation and information flow control (IIFC) inspired by the MILS approach
- A high-level solution for unified and flexible access control that can span the range of heterogeneous operating environments of a PHANTOM platform using the Next Generation Access Control standard

An initial analysis and tentative approach to both of these aspects was identified in D1.2.

Concerning the first point, component network execution integrity, the aspect that could not be addressed with ordinary operating system features was execution integrity of independent IP cores on an FPGA. We collaborated with project partners to devise an approach to this issue, to be implemented for the FPGA platform by the University of York.

To address the second point we pursued an application of the Next Generation Access Control (NGAC) standard. A vigorous effort of investigation and implementation addressing this point led to a refined view, presented in D1.3 along with a progress report and the first functional validation of a prototype NGAC implementation with feedback from the GMV use case partner. NGAC is described in 74[6].

2.7.2 Design

2.7.2.1 Design of the NGAC functional architectural approach

The NGAC functional architecture presented in the standard [6] is shown in Figure 15. PEP is Policy Enforcement Point, RAP is Resource Access Point, PDP is Policy Decision Point, PAP is Policy Access/Administration Point, PIP is Policy Information Point, and the optional EPP is Event Processing Point.

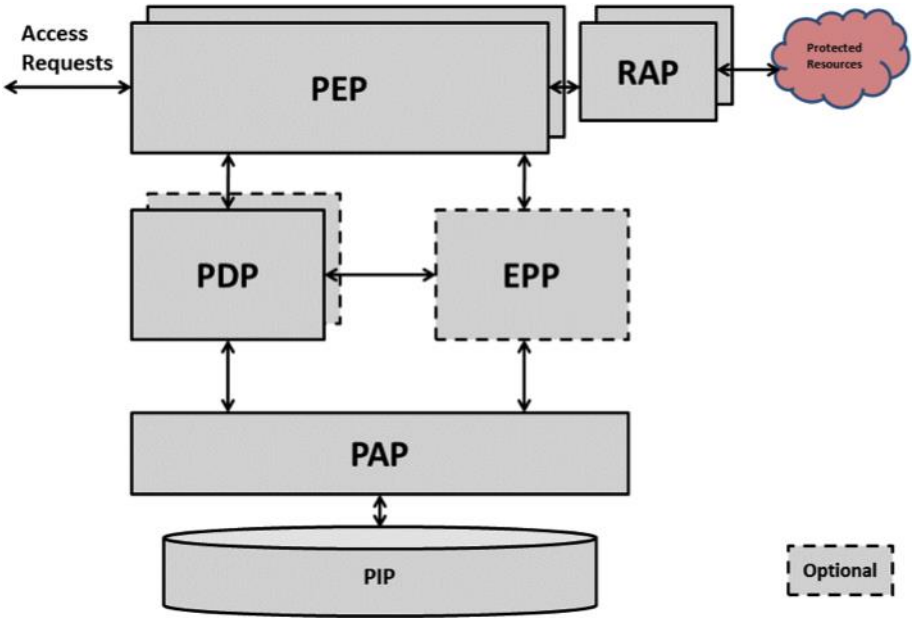


Figure 15: NGAC functional architecture per the standard

The Open Group has pursued design and implementation our own NGAC-related tools and a straightforward declarative language to express policies that comply with the NGAC framework. Specifically, a desktop command-line policy tool called 'ngac' loads policies expressed in the declarative language and can answer queries such as "access(policy1,(u1,r,o2))", the meaning of which is: "under policy 'policy1', is user 'u1' allowed to read object 'o2'?".

We have now expanded our implementation of our first simple policy tool and have designed and implemented a server with RESTful APIs for the PEP-to-PDP interface, which we call the Policy Query Interface. The PEP only needs to call the PDP through this interface with an access query that the PDP answers with "permit" or "deny". Based on this response the PEP must not perform the access to the RAP (if "deny"), or proceed to perform the object access and return the result to the application. The PEP is fundamentally a trivial decision statement conditioned by the PDP's response that performs the access on one path, or reports an error on the other path.

As part of the design effort of this project we investigated the feasibility of implementing a portable server to include the PDP/PAP/PIP functions. Through design and experimental implementation we concluded that a full heavyweight implementation of a portable server based on a database management system for the PIP, as was the case for a reference implementation of the standard, would be too costly for the project. However, we did discover that we could develop a functional lightweight and portable implementation, the design of which is described in the following.

2.7.2.2 Design of the NGAC Declarative Language

The declarative language representation is easily constructed from a graphical representation of a policy. The present declarative language does not support the entire NGAC policy framework, lacking prohibitions and obligations though it is our ambition to add them incrementally in the future as need may require.

The declarative policy specification language is defined by the following grammar.

A declarative policy specification is of the form:

policy(<policy name>, <policy root>, <policy elements>).

where,

<policy name> is an identifier for the policy definition

<policy root> is an identifier for the policy class defined by this definition

<policy elements> is a list [*<element>*, ... , *<element>*]

where each *<element>* is one of:

user(<user identifier>)

user_attribute(<user attribute identifier>)

object(<object identifier>)

object(<object identifier>, <object class identifier>, <inh>, <host name>, <path name>, <base node type>, <base node name>)

object_attribute(<object attribute identifier>)

policy_class(<policy class identifier>)

assign(<entity identifier>, <entity identifier>)

associate(<user attribute id>, <operations>, <object attribute id>)

where *<operations>* is a list:

[*<operation identifier>*, ... , *<operation identifier>*]

connector('PM')

The initial character of all identifiers must be a lower-case letter or the identifier must be quoted with single quotes, e.g. *smith* or *'Smith'* (identifiers are case sensitive so these examples are distinct). Quoting of an identifier that starts with a lower-case letter is optional, e.g. *smith* and *'smith'* are considered the same.

Additionally:

< *inh* > can be **yes** or **no**.

< *host name*> contains the name of the host where the corresponding file system object resides.

< *path name*> is the complete path name of the corresponding file system object.

Examples of policies expressed in the NGAC framework and in the declarative language, including an example from the GMV use case, along with runs of the policy tool are presented in 6.

2.7.2.3 Design of the NGAC Policy Tool

The 'ngac' policy tool for doing standalone policy development and testing has been extended with the ability to start the policy server. The policy tool offers a command line prompt, "ngac>", in response to which it accepts the following commands:

- `version.`
  Display the current version number.
- `versions.`
  Display past versions with description and current version.
- `help.`
  List the commands available in the current mode.
- `help(` <command name> `).`
  Give a synopsis of the named command.
- `admin.`
  Switch to admin (normal) user mode.
- `advanced.`
  Switch to advanced user mode.
- `import_policy(` <policy file> `).`
  Import a declarative policy file and make it the current policy.
- `newpol(` <policy name> `).`
  Set a policy to be the new current policy.
- `combine(` <policy name 1>`,` <policy name 2>`,` <combined policy name> `).`
  Combine two policies to form a new combined policy with the given name.
- `access(` <policy name>`,` <permission triple> `).`
  Check whether a permission triple is a derived privilege of the policy.
- `proc(` <procedure name> `[, verbose]).`
  Run the named command procedure, optionally verbose.
- `proc(` <procedure name> `[, step]).`
  Run the named command procedure, optionally pausing after each command.

- `script(` <file name> `[, verbose]).`
  Run the named command file, optionally verbose.
- `script(` <file name> `[, step]).`
  Run the named command file, optionally pausing after each command.
- `server(` <port> `).`
  Start the ngac server on the given port number.
- `echo(` <string> `).`
  Print the argument string, useful in command procedures.
- `nl.`
  Print a newline, useful in command procedures.
- `regtest.`
  Run built-in regression tests.
- `halt.`
  Exit the tool.

Additional unpublished commands are supported by the tool to facilitate development and diagnostic tests. The synopsis of all commands can be obtained by entering the `help` command with no argument.

The design of the 'ngac' policy tool is comprised of the following modules:

- ngac – top level module of 'ngac' policy tool; entry point and initialisation
- param – global parameters
- command – command interpreter and definition of the 'ngac' commands
- common – simple predicates that may be used anywhere
- pio – input / output of various policy representations
- policies – example policies used for built-in self-test
- test – testing framework for self-test and regression tests
- procs – stored built-in 'ngac' command procedures
- spld – security policy language definitions

2.7.2.4 Design of the NGAC Policy Server

The policy server comprises the Policy Decision Point (PDP), the Policy Administration Point (PAP), and the Policy Information Point (PIP) of the NGAC functional architecture as shown in [6]. The policy server implements a Policy Query Interface API to be queried by PEPs, and a Policy Administration Interface API to be used to establish the policy that the

server is to use to compute access queries. The latter interface was part of the PQI in the initial design, however, since policy administration is not done by a PEP it was decided to create a distinct interface.

The policy server is initiated from a shell command line interface with the command 'ngac-server', which supports several command line options:

- `--deny`, `-d` respond to all access requests with `deny` (used for testing)
- `--grant`, `-g` respond to all access requests with `grant` (used for testing)
- `--load`, `-l` <policyfile> import the policy file on startup
- `--port`, `-p` <portnumber> server should listen on specified port number

The policy server may also be initiated from within the 'ngac' policy tool by issuing the command `server( <port> ).` at the tool's command prompt "`ngac>`".

In addition to its command line options the 'ngac-server' provides two external interfaces:

- Policy Query Interface – implemented as RESTful APIs
- Policy Administration Interface – implemented as a command line interface in the policy tool and as a RESTful API in the policy server

pqapi/access – test for access permission under current policy

Parameters
- user = <user identifier>
- ar = <access right>
- object = <object identifier>

Returns
- "grant" or "deny" based on the current policy
- "no current policy" if the server does not have a current policy set

Effects
- none

pqapi/getobjectinfo – get object metadata

Parameters
- object = <object identifier>

Returns
- "object=<obj id>,oclass=<obj class>,inh=<t/f>,host=<host>, path=<path>,basetype=<btype>,basename=bname>"

Effects
- none

The ngac-server offers the following APIs as the Policy Administration Interface. A "failure" response is typically preceded by a string indicating the reason for the failure.

paapi/getpol – get current policy being used for policy queries

Parameters

- none

Returns

- <policy identifier> or “failure”

Effects

- none

paapi/setpol – set current policy to be used for policy queries

Parameters

- policy = <policy identifier>

Returns

- “success” or “failure”
- “unknown policy” if the named policy is not known to the server

Effects

- sets the server’s current policy to the named policy

  When the current policy is set to the distinguished policy name “all” then all of the policies currently loaded into the server are taken into account when an access request is computed. Of the applicable loaded policies at least one policy must permit the access and no policy may deny the access. Policies are inapplicable to a particular request if the user (or session identifier as a proxy for a user) or the object named in the access request does not occur in the policy, in which case the access request does not fall within the jurisdiction of that policy. This unique policy composition approach was developed collaboratively with the developers of the PHANTOM Repository and other Managers.

paapi/add – add an element to the current policy

Parameters

- policy = <policy identifier>
- policyelement = <policy element> only user, object, and assignment elements as defined in the declarative policy language; restriction: only user to user attribute and object to object attribute assignments may be added. Elements referred to by an assignment must be added before adding an assignment that refers to them.

Returns

- “success” or “failure”

Effects

- The named policy is augmented with the provided policy element

paapi/delete – delete an element from the current policy

Parameters

- policy = <policy identifier>
- policyelement = <policy element> permits only user, object, and assignment elements as defined in the declarative policy language; restriction: only user-to-user-attribute and object-to-object-attribute assignments may be deleted. Assignments must be deleted before the elements to which they refer.

Returns

- “success” or “failure”

Effects

- The specified policy element is deleted from the named policy

paapi/load – load a policy file into the server

Parameters

- policyfile = <policy file name>

Returns

- "success" or "failure"

Effects

- stores the loaded policy in the server
- does NOT set the server's current policy to the loaded policy

paapi/unload – unload a policy from the server

Parameters

- policy = <policy name>

Returns

- "success" or "failure"
- "unknown policy" if the named policy is not known to the server

Effects

- the named policy is purged from the server
- sets the server's current policy to "none" if the unloaded policy is the current policy
- all currently loaded policies are unloaded if the given policy name is "all"

paapi/initsession – initiate a session for user on the server

Parameters

- session = <session identifier>
- user = <user identifier>

Returns

- "success" or "failure"
- "session already registered" if already known to the server

Effects

- the new session and user is stored

paapi/endsession – end a session on the server

Parameters

- session = <session identifier>

Returns

- "success" or "failure"
- "session unknown" if not known to the server

Effects

- the identified session is deleted from the server

An active session identifier may be used as an alternative to a user identifier in an access query made to the Policy Query Interface. This correspondence is established by an `initsession` API invocation and is terminated by an `endsession` API invocation.

The Policy Administration Interface intended to be accessible only to an administratively authorised user through the policy administration tool or a process with the similar authorisation, and some functions such as `load/unload/setpol` should be accessible

only to the shell program that executes the NGAC client application. In this way, the shell that controls execution of the application would also determine the user/session and policy under which the application should execute and inform the server through the PAI.

The design of the lightweight server is comprised of the following modules:

- server – HTTP server and definition of the policy query API
- sessions – maintains correspondence to a user for each active session

2.7.2.5 Integration with PHANTOM managers

The integration of the PHANTOM Repository Server with the PHANTOM Security Framework is based on the definition of the **access domain** in metadata of the uploaded contents to the Repository and the **user-id**. Figure 16 shows an example of the metadata of two files uploaded to the Repository.

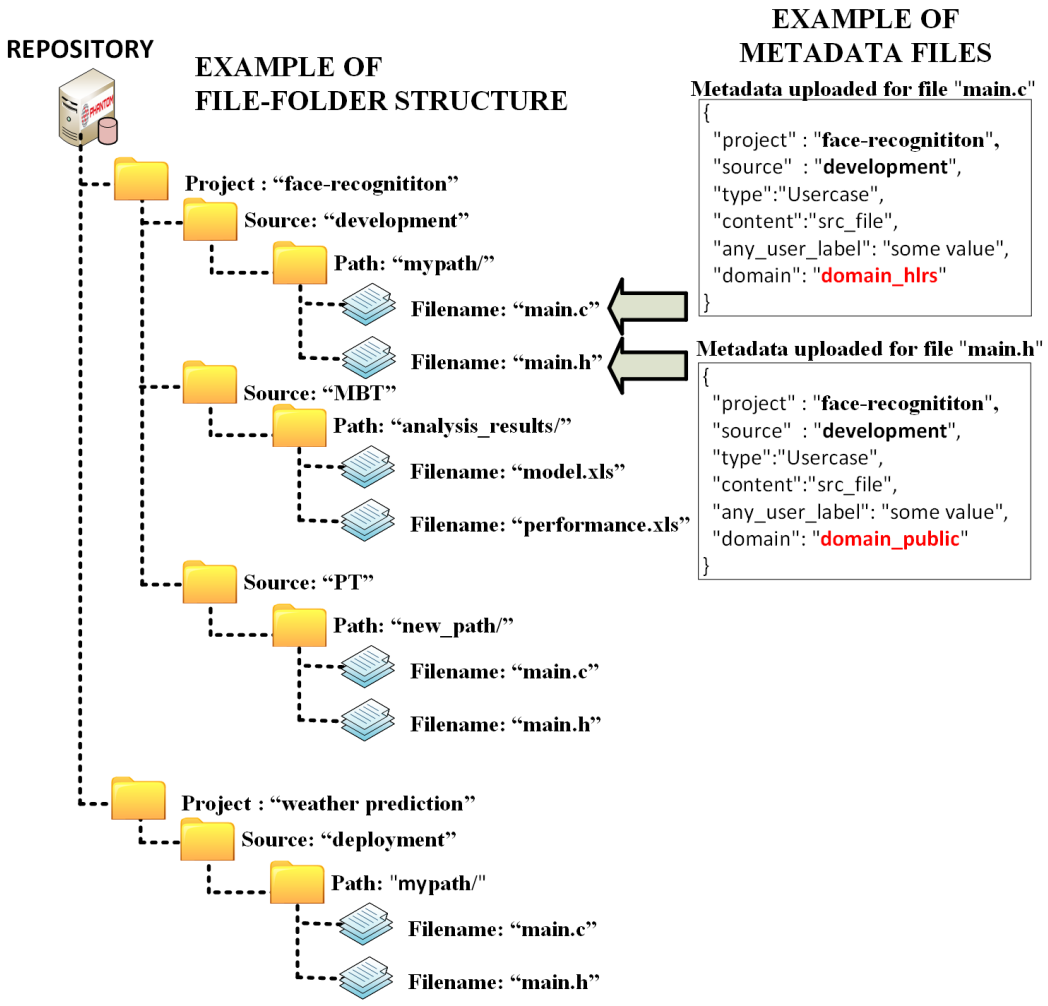


Figure 16: Example of metadata of two uploaded files in the PHANTOM Repository

The Repository Server requests authorization to the PHANTOM Security Framework each time a user wishes to download from the Repository. For such requests, the Repository

needs the **user_id** which is obtained from the token authentication used by the user, and the **domain** of the file defined in its metadata. Following is an example of requesting authorisation for "**read**" access by the user "**bob@abc.com**" for a file with domain "**domain_hlrs**".

```
curl –XGET http://127.0.0.1:8001/pqapi/access?user=bob@abc.com&ar=read&object=domain_hlrs
```

The Security Server grants the access when the security policy includes the user_id in the domain of file. The response is the string "**granted**" when effectively the user is included in the file domain, or the string "**denied**" in the other case.

Figure 17 shows an example of the definition of two private domains and the public domain policies for the PHANTOM Repository. In the example four users are defined, two belonging to each private domain.

```
policy('Policy (phantom_demonstration)','File Management', [
    /** Define the users */
              user('user1@hlrs.de'),
              user('user2@hlrs.de'),
              user('user3@opengroup.org'),
              user('user4@opengroup.org'),
    /** Define the type of objects in the repository*/
              object('domain_public'),
              object('domain_hlrs'),
              object('domain_opengroup' ),
    /** Assign the users to global set */
              user_attribute('Users'),
              user_attribute('Users_hlrs'),
              user_attribute('Users_opengroup'),
    /** Assign the users to groups */
              assign(user1@hlrs.de','Users_hlrs'),
              assign('user2@hlrs.de','Users_hlrs'),
              assign(user3@opengroup.org','Users_opengroup'),
              assign('user4@opengroup.org','Users_opengroup'),
    /** groups */
              assign('Users_hlrs','Users'),
              assign('Users_opengroup','Users'),
    /** Assign all the attributes to the class 'File Management' */
              policy_class('File Management'),
              connector('PM'),
              assign('Users','File Management'),
              assign('domain_public','File Management'),
              assign('domain_hlrs','File Management'),
              assign('domain_opengroup','File Management'),
    /** define the permitted operations of each group to each type of object */
              operation(r,'File'),
              operation(w,'File'),
              associate('Users',[r,w],'domain_public'),
              associate('Users_hlrs',[r,w],'domain_hlrs'),
              associate('Users_opengroup',[r,w],'domain_opengroup')
      ]).
```

Figure 17: Example policy for the PHANTOM Repository

2.7.2.6 Design of Execution Integrity

We now describe the design of the execution integrity aspect of the PHANTOM Security Framework design.

This aspect of the PHANTOM platform security solution underlies *all* PHANTOM application execution, and provides a foundation for component network integration and for higher-level application-specific security mechanisms. All PHANTOM applications, whether embedded or enterprise scale, benefit from the low-level execution integrity controls of the PHANTOM platform.

A PHANTOM platform is constructed from a number of potentially heterogeneous hardware and software components. Borrowing terminology from MILS, the components of the platform are referred to generally as *foundational components*. For a platform comprised of a CPU, a GPU, and an FPGA, there would be three corresponding foundational components, each consisting of its hardware, firmware, and/or software.

The components making up an application specified by a developer to achieve a particular purpose is referred to as a *component network* that specifies the permitted interactions among those components. Together the components and their interactions may be represented as a graph the interaction relations of which are enforced by the PHANTOM platform.

Our design analysis has led to the conclusion that current GPUs are limited and must be augmented by a CPU to manage its use in a periods processing (serial) fashion with intervening flushing of storage. FPGAs on the other hand do present the possibility of achieving isolation for IP cores like that of an operating system. CPUs of course are already managed in an appropriate way by the operating system under which they are running.

Execution Integrity of a Component Network

The PHANTOM programming model provides the ability to construct an application that is a composition of communicating, cooperating components referred to as a *component network*. The component network is a modular construction of application or PHANTOM components (operational components) that enables component reuse and reasoning about the operation of the component network based on the known functional attributes of the components and the manner in which they are composed in the component network.

When reasoning about the composition of components in a component network a critical *assumption* is implicitly made: that the execution environment guarantees that the abstraction of the component network is faithfully refined in its realization. Specifically, that the distinctness of components represented in the component network, the unique source and destination of each inter-component communication connection, and the absence of interference among components, or upon components and their communications by other parts of the system, are preserved in the execution environment provided by the PHANTOM platform.

The properties of the platform that provide the basis for the component network assumption are: *isolation* and *information flow control* (IFC). That is, the platform as a whole, and each node (hardware processing element along with its control firmware and software), must guarantee that each exported resource (process, memory, etc.) is free from undefined external interference (isolation), that only defined (by the component network) inter-component interference may occur through communication mechanisms provided by the platform, and that such communications are free from interference from other sources (IFC).

The ability of the PHANTOM platform to provide isolation and IFC to a component network is based on the ability of each of its nodes to provide isolation and IFC for the resources it exports that are involved in the component network. Hence, we provide an account for the manner in which each type of node in the heterogeneous PHANTOM platform provides isolation and IFC for the resources it exports. Together, we refer to the combined effect of these properties of isolation and IFC of the separate processing elements as *execution integrity* of the component network.

Execution Integrity of CPU Processes

Generally, PHANTOM software components run as, or are supported by CPU processes. These processes are created by the kernel of a general-purpose operating system (OS) such as Linux, or a real-time operating system in certain applications. The isolation of software components relies on standard Linux kernel process isolation. Each process is run as an ordinary user with limited privileges to avoid any potential interference. Some system processes are run as special system user having greater privileges.

Part of the service provided by the OS is the management of the primitive resources that are shared among the processes being run by the OS, including main memory, CPU cycles, and hardware devices that are integrated with the CPU such as the system clock or a GPU. Other hardware devices, considered add-on peripherals, such as mass storage devices and network devices, are controlled by driver software and associated subsystems such as file systems and network stacks. These trusted software systems in turn provide separation among the abstract resources associated with the primitive resources of these devices as they are being used by different users and processes acting on behalf of those users.

In the case of CPU processes the OS and the hardware on which it runs are *trusted components* of the platform. From the standpoint of (application) component network execution integrity these platform (foundational) components are trusted to correctly provide isolation and IFC for (application) software components running in CPU processes.

Execution Integrity of GPU Processes

Application component code running on the Linux kernel is afforded isolation and interference protection by the standard mechanisms in the kernel. GPU drivers attempt to provide similar mechanisms, by grouping memory into contexts. A context is owned by an application process and may only be read by that process. Memory on the device is allocated to a context by the GPU drivers, and it cannot be accessed by other processes unless explicitly declared as shared memory. For most use cases this is sufficient.

GPU processes are potentially susceptible to memory leakage and object reuse problems. Unfortunately, these problems make it difficult to reason about security in the GPU domain. First, driver code to interact with the device is both proprietary and secret. This means that we cannot perform the security audits that we can with open source code, and we cannot edit the code to improve it with attack mitigation techniques. Second, the focus of GPU drivers is on performance first, and security second (or not at all). Unlike a standard OS, most GPU implementations do not zero memory that is allocated to a process, meaning that it may contain data from a previous user. Even worse, GPUs make it difficult to erase the entirety of your own memory, as one can do on a standard OS process, because of the existence of constant memory and other optimisation-focussed limitations. There are many live attacks

against current GPU implementations, and little that can be done by anyone other than the manufacturer to prevent them. If high security is required then it is recommended that one of the following options be taken:

1. Avoid the use of the GPU from the process to be secured from exfiltration. FPGAs can be hardened with confidence so should be considered a better option.

2. If the GPU is to be used, ensure that the only process using it is the one that is being secured, and prevent all other processes from accessing the GPU.

Execution Integrity of FPGA IP Cores

In most systems, hardware is trusted with full system access. However, in the PHANTOM environment FPGA hardware designs come from the user and cannot be trusted, so it is necessary to ensure that hardware components can access only the memory and devices associated with their software part, and nothing else. If any component mapped to the FPGA infrastructure requires security isolation then additional capabilities are automatically added to the infrastructure to ensure that malicious IP cores cannot affect system integrity, and that cores handling sensitive data are protected from observation by other cores. We present here a summary of the approach taken; details are provided in deliverable D4.3.

A PHANTOM FPGA component mapped to an FPGA consists of a hardware part and a software part. The software part exists as a standard Linux process, and the hardware part is an IP core implemented on the reconfigurable logic fabric of the FPGA. This provides a unique security challenge, because it is essential that the software part of a component can communicate with its hardware part and vice versa, but it must not be possible for the software to communicate with the hardware from a different component.

The isolation of software components relies on standard Linux kernel process isolation. As noted above, each process is run as an ordinary user with limited privileges to avoid potential interference. Hardware components are controlled from the ARM cores using memory-mapped registers in the CPU's standard address space, as well as including a portion of shared memory for each component that both the hardware and software can access. Hardware components are mapped into the Linux kernel's UIO device space, with a device node being created for each device. The permissions of this node are then set to ensure that only the correct corresponding software processes can access them. This is automatically implemented using the Linux device manager udev, and enforced by the kernel.

To ensure that hardware components can only access the memory to which they are permitted, a minimal memory management unit (MMU) is added to the memory connection of each hardware device. Figure 18 depicts the AXI SmartConnect MMU used to restrict hardware access to memory on FPGA devices. The MMU is preprogrammed by the system infrastructure and cannot be changed by the system at runtime, thereby isolating components into their own memory spaces.

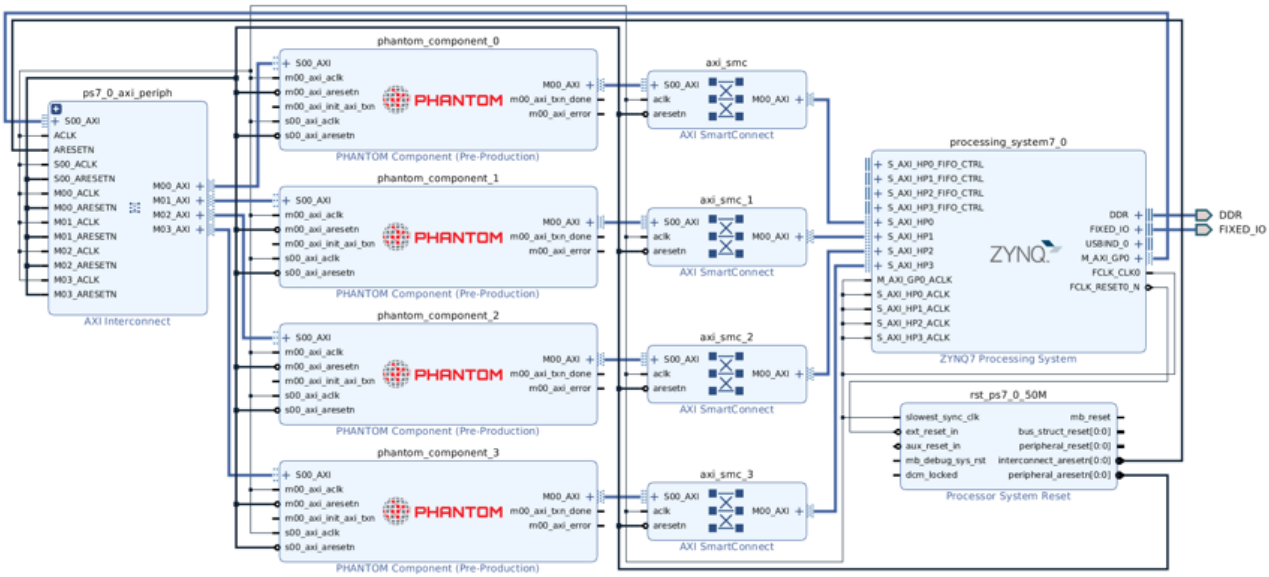


Figure 18: AXI SmartConnect MMU restricts memory access on FPGA devices

The AXI bus is not a multi-drop bus, but rather a set of point-to-point connections. Reads and writes made by one IP core are not visible to other cores. The structure and address ranges of the bus are all defined statically at architecture-build time. Each bus endpoint has a range of addresses that it responds to, so that at runtime a read of any given address will always go to a specific location and no others. These ranges cannot overlap or the bus fails. Endpoints filter out transactions for addresses they are not responsible for, and this filtering is done statically by the constructed architecture, not by any user-accessible logic in the IP cores. Thus, the solution provides full protection.

2.8 MODEL-BASED TESTING

In PHANTOM, Model-based Testing (MBT) conducts functional and non-functional verification and validation for PHANTOM applications. The MBT design allows testing very early in the development life cycle of the applications and then execute concrete test cases against applications for more detailed validation in a later stage.

MBT in PHANTOM is designed as four testing activities - model validation, performance estimation, functional testing and non-functional testing - in two main phases - early validation and test execution. Figure 19 lists the MBT activities in PHANTOM and the alignment with PHANTOM stages.

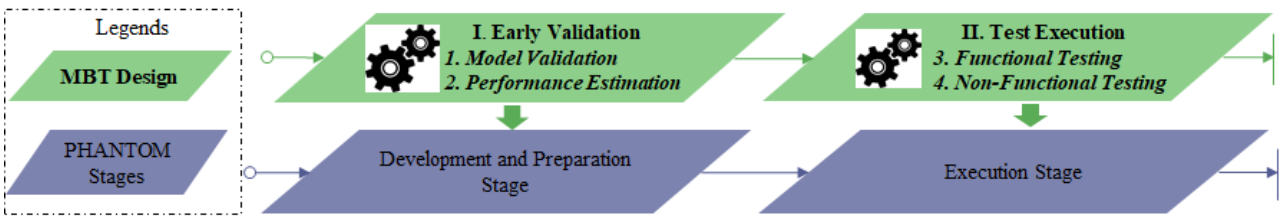


Figure 19: Model-based Testing design for PHANTOM verification and validation

Generally, early validation is the first MBT phase in PHANTOM, which tests and validates the design of an application's functional and non-functional properties. Early validation only relies on design specifications but does not require the execution of applications, and it is an early testing phase in parallel with the application development to detect design defects for follow-up correction and prevent them from escalating to implementation. Test execution is the second MBT phrase in PHANTOM to execute concrete test cases against the SUT and collects thorough testing results for both functional and non-functional properties. Testing results are sent back to developers for further analysis and improvement. During the whole test execution phrase, traceability is kept between SUT specifications, test cases and testing results, so that developers can easily trace the defects sources from testing results to design specifications, and correct the defects based on the testing results. The details of each MBT activity is detailed below.

2.8.1 Model Validation

Model Validation simulates the MBT models created following design specifications to check if the intended implementation of applications contains any deadlock or overdesign meaning that certain functions are never used. MBT models are created following application specifications and represent an abstraction of the to-be-implemented application. Without executing the applications, simulating the corresponding MBT models produces a functional analysis. The model validation workflow is illustrated in Figure 20.

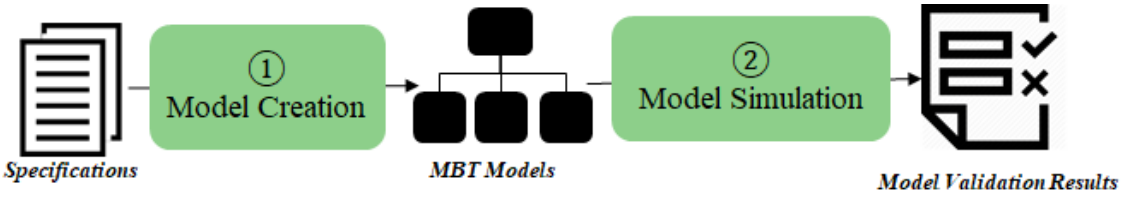


Figure 20: Model validation workflow

In all figures in section 2.8, the green rectangles represent MBT steps in one activity. The start and end point of an arrow represents inputs and outputs of this step, and the start and end point of the whole workflow is the global inputs and outputs of this MBT activity.

Step 1. Model Creation. MBT models are developed following use case design specifications for validation purpose. The specifications define the testing requirements or the aspects to test of SUT (e.g., functions, behaviours and performances). In PHANTOM, communicative state machine is used as the meta model for the creation of MBT models to consider the inner communication of application components, and the model is created in xLIA language (eXecutable Language for Interaction & Assemblage) [1] due to the variety of primitives and the support of encoding all classical semantics.

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

The model simulation is achieved by DIVERSITY [2], an open-source Eclipse based tool for formal analysis. We use DIVERSITY for model simulation purpose due to its algorithm of symbolic execution. Symbolic execution uses symbolic parameters to represent simulation inputs rather than concrete numerical values so that multiple scenarios can be evaluated at the same time to simulate the models and explore the model behaviours more efficiently.

2.8.2 Performance Estimation

Performance estimation estimates the performance of newly designed applications (e.g., execution time, energy consumption) reusing existing and previously tested components and composes them in a different way to achieve different functional objectives. Once a new application is designed reusing the existing components, performance estimation gets the component network description of newly designed application and deduces the global performance of the newly designed application by analysing the patterns of how components are composed together and previously collected performance metrics of each component. The performance estimation workflow is illustrated in Figure 21.

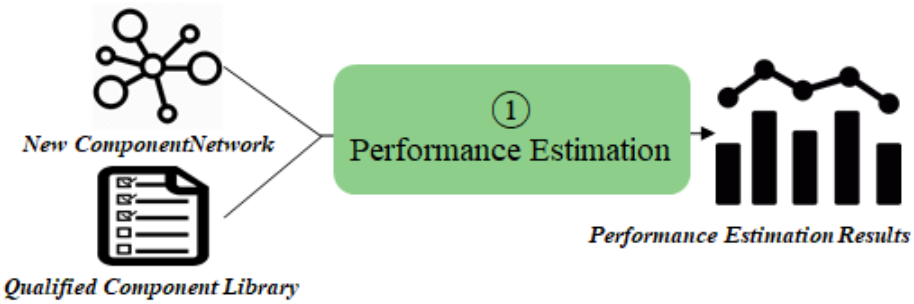


Figure 21: Performance estimation workflow

Step 1. Performance Estimation. When given a component network description of an application, this step identifies the composition patterns among inner components, estimates the performance of the application based on an estimation model related to each non-functional property, and provides the estimation results. We have identified four types of composition patterns (i.e., sequence, parallel, condition and iteration) to represent the composition relations among application components. They cover most of the structures specified by workflow languages or workflow patterns [3]. For each performance property, we have defined an estimation model as shown in Table 1 to calculate the application's performance based on their individual components' performance and composition patterns. This is particularly useful to estimate the performance of applications that reuse existing components and compose them in different ways to complete other computing tasks, such as HPC to simulate different topological structures with same components.

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

Property Name	Estimation Methods for Composition Patterns			
	Sequence	Parallel	Condition	Iteration

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

2.8.3 Functional and Non-Functional Testing

Functional testing stimulates the system under test (SUT) with different input data, executes the SUT and obtains the output data. Functional testing then compares the observed output with the expected output to decide if the functions of SUT are correctly implemented. Non-functional testing shares the same workflow with functional testing, and at the end of execution, performance information is collected for two purposes: 1) to generate test verdicts indicating if non-functional requirements are satisfied, 2) to collect performance metrics of application components for MOM as mapping references. Both functional testing and non-functional testing shares the same workflow as presented in Figure 22. However, since the functional testing and non-functional testing have different testing aspects, the MBT models and the executed test cases for the two activities are different from one another.

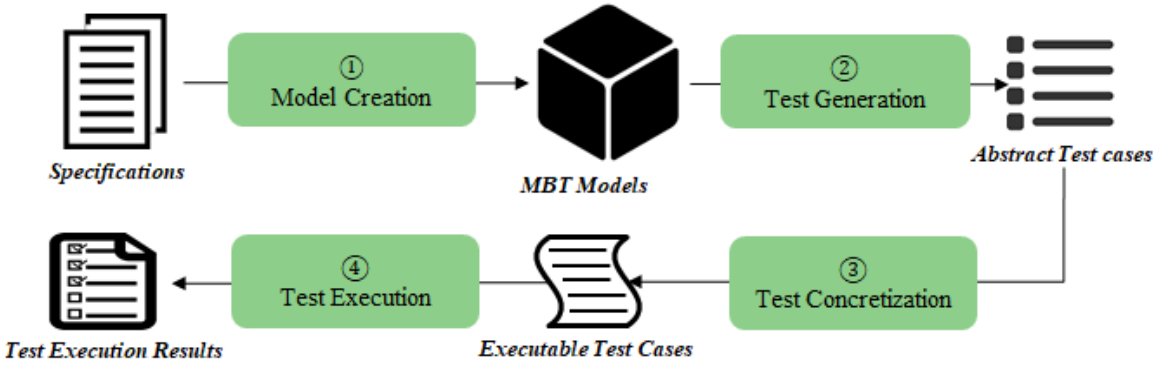


Figure 22: Functional testing and non-functional testing workflow

Step 1. Model Creation. MBT models are created following design specification for test generation purpose. The MBT models are developed in communicative state machine for each use case in xLIA language.

The MBT models for test generation are different from the MBT models for validation in the following two points. Firstly, the MBT models for test generation do not only include the SUT's functional aspect of dynamic behaviours, but also contain the non-functional testing state and transitions to collect performance information. Secondly, the MBT models for test generation focus on global inputs and outputs, which are different from MBT models for model validation with internal control and data details. This is because

in order to provide a comprehensive validation result, the MBT models in early validation phase are expected to contain both internal data/control flows of components within an application as well as the global input-output relations between an application and its environment. Since the early validation is in parallel with development and does not impose timing constraints, a detailed model is important to largely exploit the model behaviours and detect early stage defects. Conversely, the MBT models in the test execution phase for test generation contains necessary input-output relations, as only inputs and outputs information are used to decide the pass/fail of a test, while internal communication is not captured during application execution, which also helps to improve testing efficiency.

Step 2. Test Generation. The second step automatically generates abstract test cases from MBT models by applying the test selection criteria to test generation tool. In PHANTM, we use the DIVERSITY tool for test generation. Besides the symbolic simulation algorithm presented in model validation section, DIVERITY is able to generate test case in TTCN-3 languages [4]. TTCN-3 is a standardized testing language developed by ETSI (European Telecommunication Standards Institute), with the advantages of multi-purpose testing support compared to other testing languages such as real-time support and distributed execution support, which highly align with the testing requirements and challenges in PHAMTOM distributed embedded environment. The transition coverage is used as the selection criteria to generate test cases in order to cover all transitions and test all aspects in design specifications. It worth mentioning an additional TTCN-3 is designed and developed within PHANTOM to correct grammar errors in the test cases generated by DIVERSITY, while the details is introduced in D3.2.

Step 3. Concretization of Test Cases. The third step concretizes the abstract test cases from step 2 to executable test cases with mappings between the abstraction in MBT models and system implementation details. Concrete test cases contain low-level implementation details and can be directly executed against the SUT. The Codecs/Decodecs are developed in PHANTOM in TTCN-3 to support the concretization of test cases for three use case applications.

Step 4. Execution of Test Cases. The executable test cases are automatically executed against the SUT (system under test). During the execution, the SUT is provided with inputs from each test case, and the outputs (for functional testing) and performance information (for non-functional testing) are collected to generate test verdicts. System adapters are developed to provide communication channels to automatically execute test cases by connecting SUT with the test execution environment and data exchange.

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

2.9 RESOURCE, APPLICATION, EXECUTION MANAGERS

The Resource, Application and Execution managers are lightweight services, with web and restful interfaces, which registers and support the progress of the PHANTOM toolset. This section describes their final design, architecture and their functionalities.

The Resource Manager (RM) stores information of the available hardware, the Application Manager (APPM) stores the progress of the registered applications through the different stages of the PHANTOM toolflow, and the Execution Manager (EXM) keeps registered and summarized statistics of the different executions of the applications. Additionally, all of these managers provide a pub/sub (publish-subscribe) mechanism for PHANTOM components to subscribe to changes of particular fields of their information and get notified when be updated. For instance, the RM, APPM, and the EXM can notify when there is a change on the status of a particular device, status of a particular application, or execution of a particular application, respectively.

Each one of these managers is composed of a data storage layer, used to persistently store their respective data, and a web-service for the data transmission from the agents to the data storage layer. Figure 23 shows the software framework and the interfaces of the Managers.

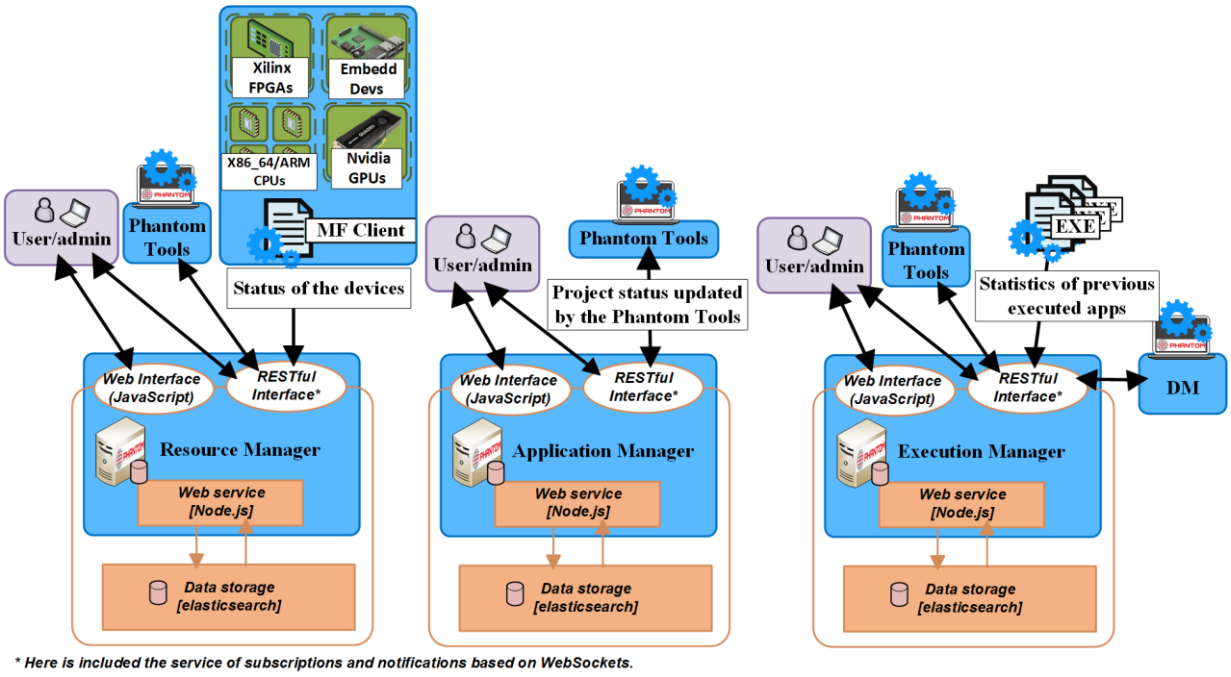


Figure 23: Software frameworks and interfaces of the PHANTOM Managers: (a) Resource Manager, (b) Application Manager, and (c) Execution Manager

The Web service is written in JavaScript under the Node.js runtime environment. Node.js uses an event-driven architecture and is suitable to be used for data-intensive real-time applications that run across distributed devices. A free and open-source framework for Node.js – Express.js is used to build the RESTful APIs.

For the data storage component, ElasticSearch is used. It is a flexible and powerful open-source, real-time search and analytics engine. As a distributed, multi-tenant full-text

search engine, it is preferred as supporting RESTful web interface and using schema-free JSON documents.

2.9.1 Resource Manager

The Resource Manager (RM) is a lightweight service which provides information related to the available heterogeneous infrastructure. The Resource Manager (RM) performs the following tasks:

- **Resource registry.** This service is responsible for maintaining the list of all heterogeneous devices of the common infrastructure. The service provides interfaces for adding/editing/deleting devices to or from the infrastructure. Each entry of the resource registry contains information about the resources such as the network IP address, type of processors, amount of memory, and installed accelerators (GPU or FPGA), static configuration information (like number of cores), as well as some additional fields helpful for the identification of the devices (ID, name, MAC-address, or IP address). The register of devices is only done when a new device is added or updated to/in the system.

- **Resource status tracking.** This service provides online data about the current status (CS) of all registered infrastructure resources. Utilization of computation cores, available RAM, actual I/O and memory bandwidth as well as other essential information on the resources are included. These data is constantly updated by the Monitoring Clients running on the devices. Notice that the MF-Server keeps all the monitored metrics received, while the RM only retains the most recent value of each one of the metrics.

- **Default monitoring parameters (Monitoring Configuration):** The RM stores the monitoring configuration (MC) and supplies it when requested. The MC specifies what to measure on each device, and how often to measure it.

Figure 24 shows the transferred information between the Resource Manager, users, and the PHANTOM tools.

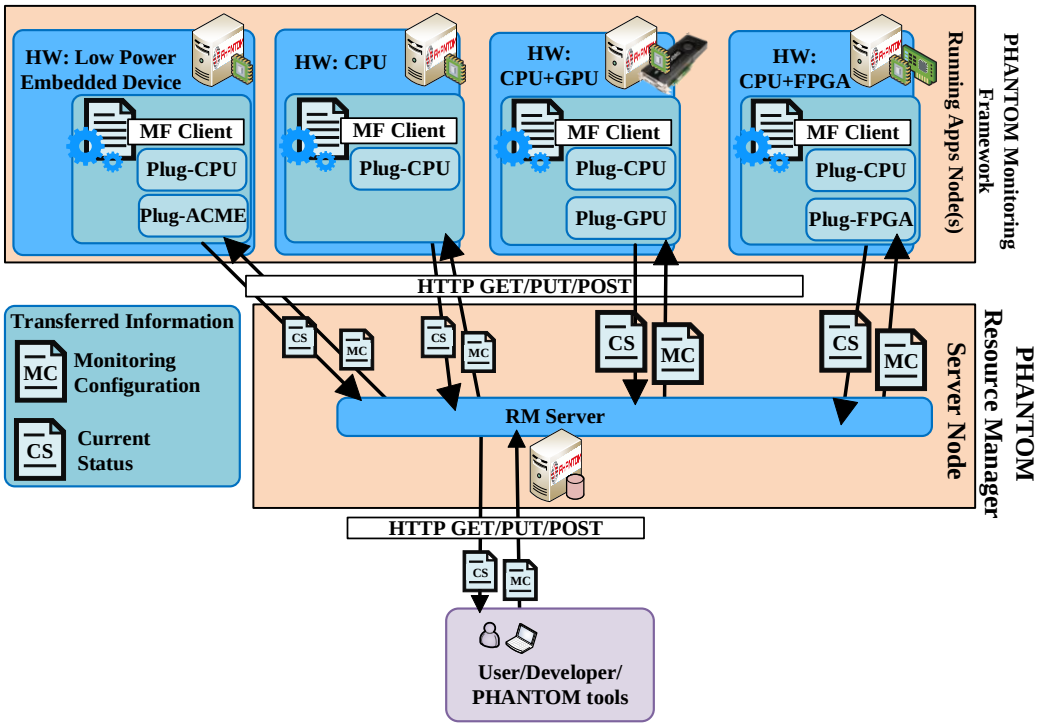


Figure 24: Interaction between Resource Manager and the users and PHANTOM tools

2.9.2 Application Manager

The Application Manager (APPM) is responsible for the following tasks:

- **Application registry**. This service allows application developers to enrol their applications into a central registry that retains the most essential information about the application such as location of source code in the Repository, availability of binaries, etc.

- **Application status tracking.** This service is responsible for acquiring the status of the application instances along the development stages.. In addition, it provides a subscription mechanism for on the updates of the tools waiting data, such as the completion of previous preparation stages. Therefore, this finalization of particular executions. The service relies on the different PHANTOM tools update their progress.

Figure 23 (b) shows the software framework and the interfaces of the Application Manager.

2.9.3 Execution Manager

The Execution Manager (EXM) is responsible for the following tasks:

- **Store execution data** (such as summarized metrics and details of which application and where it ran). The data is collected at the application level by the applications instrumented with the MF-Library. The Execution Manager provide the functionalities to query and analyse the registered data.

- **Execution status tracking.** The Execution Manager provides a subscription mechanism on the updates of the data, such as notify the finalization of particular executions. The notifications consists on sending a JSON data file which contains information as the identification of the application, the device where it was executed, the timestamp of the finalization and the total required time for the execution.

Figure 23(c) shows the software framework and the interfaces of the Execution Manager.

2.10 PHANTOM TOOLS INTERACTIONS

As is described in detail in D5.2, the PHANTOM servers work on top of the other tools to allow any necessary coordination. Summarizing the communication model displayed in Figure 25, it can be described in 3 stages:

- Static analysis: During this stage, the tools that run statically on the user's code and description files are executed. These include PT's Code Analysis, the IPCore Generator and MBT's Early Validation.
- Design Exploration: This is an iterative stage in which the MOM continuously generates new mappings according to the results of the static analysis and follows their execution using the Execution Manager. Additionally, the MBT also requests specific executions to deploy tests for individual components. The developer is also able to request deployment mappings of their own choosing.
- Building stage: During this stage, the design decisions are implemented by PT's Technique Selection and the Deployment Manager, gathering together the source files that will be used for compiling the application , deploys the components' code on the target machines and the compiles them
- Execution stage: During execution, the deployed application uses the Monitoring Library to gather useful metrics and upload them to the Monitoring Server for MOM and MBT to access them and proceed with stage two all over again.

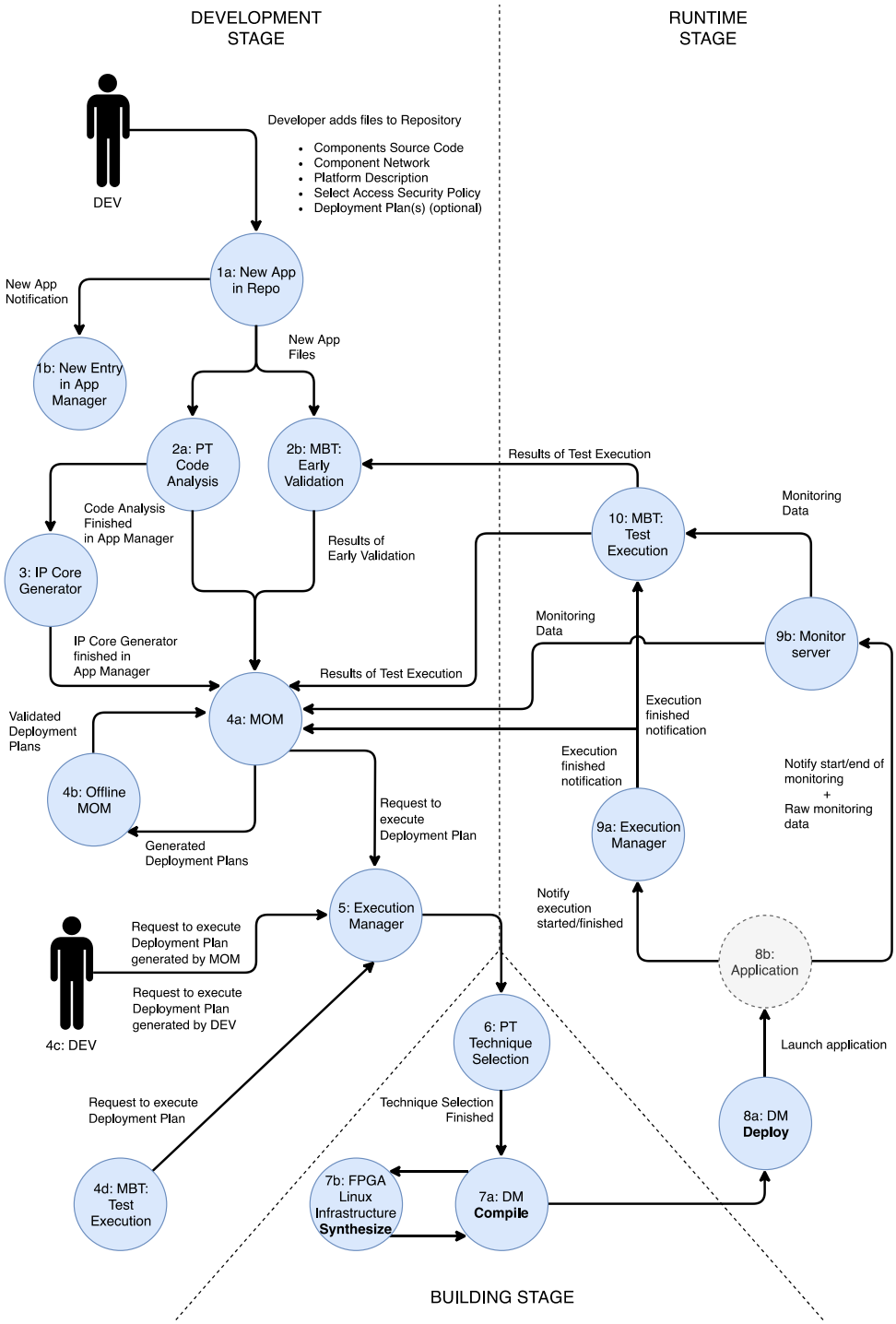


Figure 25: PHANTOM tools workflow

2.10.1 Repository

The PHANTOM Repository plays an important role in the development stage as well as in the run-time stage. It supports the exchange of information and files between the different PHANTOM tools, as well as between them and the running applications, and also among the Applications and PHANTOM tools with the users. For such purpose the Repository stores metadata (in JSON format) associated to each stored file. This eases the

interaction between them because they do not have to run simultaneously, or on the same device.

2.10.2 Parallelization Toolset

The PHANTOM Parallelization Toolset includes many interactions with the other tools via the Repository, where all the application files are stored and the Application/Execution Managers which are used for the overall coordination. In specific, it downloads the source files for their analysis along with the component network and uploads the results, while notifying the Application Manager about the end of each stage. Technique Selection waits for requested executions registered at the Execution Manager to select the corresponding versions of the deployment.

2.10.3 Multi-Objective Mapper

The MOM needs to interact with other components in PHANTOM architecture, namely the Model-Based Testing, the Parallelization toolset, the Monitoring library and framework, as well as the Repository, from which MOM receives its input and stores its outputs. The MOM requires as input the component network specification, the hardware platform specification and the user defined requirements and, if available, the initial mapping. Furthermore, MOM collects estimations on the non-functional requirements from the Model-Based Testing toolset, while results from previous application deployments on the underlying hardware platform (performance data) are derived from run-time monitoring through the Monitoring Framework. In addition, the Parallelization toolset provides MOM with parallelization directives to further assist MOM in the mapping optimization process. It then performs a number of optimization steps to generate the optimised deployment plans (mapping decisions of application components to hardware platform processing elements). All interactions that MOM performs with the rest of the PHANTOM components are facilitated and implemented through the use of the Repository. The latter provides secure interactions using a token mechanism to manage the access rights of PHANTOM users.

2.10.4 Deployment Manager

The Deployment Manager follows the status of PT's Technique Selection until the latter completes for a given execution and uploads the necessary source files on the Repository. From this point, DM deploys the execution that is requested by MOM, MBT or the user. To that end, the refined source code is uploaded on the Repository along with the compiled binaries. Control then goes to the application code which is the one to send monitoring data to the Monitoring Framework and updates the execution's status on the Execution Manager upon termination.

2.10.5 Monitoring Server

The main interaction of the PHANTOM monitoring tools are the collection of data from the Monitoring Client (monitoring on system level) and the instrumented applications (metrics at the application level), the provision of access to that data. This access to data is available for the PHANTOM tools and users through the Monitoring server API, and additionally users can inspect the data throw a graphic web-interface provided by Grafana.

2.10.6 Security

The Security Policy Server provides a Policy Query Interface through which the Policy Enforcement Points (PEPs) of the Repository and other Managers query the current policy for access request verdicts. These managers or other trusted components of the execution framework can call the Policy Server through its Policy Administration Interface to load/unload policies, combine policies, and select a policy or combination of policies to be the current policy for queries to the PQI.

2.10.7 Model Based Testing

From MBT perspective, the execution and test of an application deployed over PHANTOM system relies on the interaction of test execution system of MBT, Repository and Execution Manager, which generally can be summarized as follows:

- MBT sends to the Repository the input data and an optional deployment plan if a predefined mapping needs to be respected.
- MBT sends an execution request to Execution Manager and receives an execution id as the identifier of this execution.
- MBT subscribes to the execution status in Execution Manager.
- MBT waits until receives the notification from Execution Manager, indicating the execution is over. The notification contains the performance information related to this execution in the same message.
- MBT retrieves execution outputs (files) from Repository
- MBT generates testing verdicts based on the outputs and/or performance information.

2.10.8 Application Manager

During the development stage the Application Manager keeps register of the user/developers' applications and their status such completed the upload of source code in the Repository, completed processing on each phantom tool, and the availability of applications ready for their execution. In other words, the Application Manager supports the interaction of the PHANTOM tools and users during the development process previous to the execution of applications.

2.10.9 Execution Manager

The Execution Manager is the central server for supporting the Phantom tools interaction during the execution stage. It keeps track of the different stages of the executions, from their request up to be completed or rejected. Additionally, it collects additional summarized information, mainly the summarized statistics of the metrics and other information stored by the Phantom tools (such MBT or DM). The metrics are collected at the application level by the applications instrumented with the MF-Library and Exec-Library.
The Execution Manager server provide services of query and analyse the registered data, a subscription-notification mechanism on the updates of the status of the applications,

such as notify the finalization of particular executions. The notifications consists on sending a JSON data file which contains information as the identification of the application, the device where it was executed, the timestamp of the finalization and the total required time for the execution.

2.10.10 Resource Manager

The Resource Manager server, keeps track of the computing devices in the system as well as the availability and load of their resources. This service provides the most simple and efficient way to knowledge of the available hardware if we consider that can change dynamically and the heterogeneity of the supported hardware in the PHANTOM tool chain.

3. IMPACT OF PHANTOM TECHNOLOGIES

3.1 INNOVATIONS BEYOND THE STATE-OF-THE-ART

This section provides a short revision of the state-of-the-art in the core research and innovation areas that are targeted by the PHANTOM technology.

3.1.1 Repository

The Innovations of the PHANTOM repository are:

- **Control Access:** The Repository allows controlling the access to the files stored in it, based on a domain access definition in the metadata of the files and the authorization of the PHANTOM Security Framework.

- **Service of notification of updates.** The Repository provides a subscription mechanism on the updates of the files on a particular project/source/path. The notifications consists on sending the JSON data of the respective files as they be updated. This is useful in scenarios such as PHANTOM users trigger the execution of an application when given input data, and users are notified when the execution is over and output data is ready.

- **Flexible Metadata:** The Repository allows the PHANTOM Tools complete freedom to add their own data into the metadatada (JSON format) of each file stored on it. These ease the iteration of PHANTOM tools. Additional functionalities for the Repository can be developed to react depending the on the metadata.

3.1.2 Parallelization Toolset

Use of the PHANTOM Programming Model

The adoption of the Programming Model (specifically its enhanced version including the semantics annotations as described in D1.3) confronts some of the issues that the latest technologies in the science of automatic parallelization have yet to resolve. It enables the user to interact with the tool via high level annotations optimizing the results produced by their joint effort with the tool. This approach aims to ensure that the best techniques have been inspected with regard to the parallelization efficiency provided by the PHANTOM framework.

The Programming Model assumes an architecture where the computational work of the application is split into strictly separable software components, suitable to offloaded into hardware accelerators such as GPUs, FPGAs and/or complex non-uniform architecture such as NUMA systems with different memory topologies and characteristics.

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

IP Core Marketplace and IP Core Generator
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. 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 PHANTOM FPGA Linux 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 7cores. 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.

3.1.3 Multi-Objective Mapper

The MOM is based on a novel approach inspired by multi-objective and bio-inspired paradigms. Innovation relies on multi-dimensional optimization: against multiple objectives and targeting systems with increased heterogeneity (CPU, CPU-GPU, CPU-FPGA).

The Generic MOM does not rely on existing tools and developments, but is a development from scratch. The novel problem formulation (described in details in Deliverable 2.2) has been developed to address the idea of multi-dimensional optimization in full consistency with the PHANTOM Component-based Programming Model. Based on this problem formu-

lation, MOM has been developed using Java Programming language and Eclipse platform as Integrated Development Environment, using appropriate XML libraries to be able to parse its input and produce its output. As MOM is inspired by multi-objective optimization and bio-inspired paradigms, several objectives can be considered. The genetic algorithm which is developed is a metaheuristic inspired by the process of natural selection and which relies on bio-inspired operations such as mutation and selection. Generic MOM implementation consists of a sequence of steps executed in iteration, which have been implemented thoroughly in the scope of PHANTOM and optimized for addressing PHANTOM requirements.

The offline analysis work does not intend to create new analysis techniques. The primary innovation in this approach is the seamless integration of the state-of-the-art into the development process in a way that allows the value of that research to be used by non-experts. The platform automatically makes use of the offline MOM to reject failing mappings early, thereby saving development time. This does not require interaction with the user.

The Component-based Programming Model was defined in order to assist with real-time analysis. By requiring applications to be componentised and their communications explicitly elaborated, it is possible to analyse the system in parts. This is an advance over traditional real-time analysis, which attempts to consider the entire system as a whole. This still relies on a communications platform/protocol that has determinable worst-case performance guarantees.

3.1.4 Monitoring Framework

The innovations of the PHANTOM Monitoring Framework are:

- **Extended and Improved design of ATOM.** The PHANTOM Monitoring Framework architecture follows the design of ATOM – the monitoring solution elaborated by the EU EXCESS and DreamCloud projects. However, the initial design of ATOM was too infrastructure-oriented and application-level monitoring was not supported. Therefore, ATOM has been considerably redesigned to enabling application-level monitoring, support heterogeneous hardware resources (including hardware accelerators as Nvidia GPUs and series 7 of Xilinx FPGAs), and use a more modular component-based and service-oriented architecture, for enabling component-based organization of the PHANTOM applications. Usability and configurability on the user-side is also improved.

  In addition to the support of hardware accelerators, it was added **support for external Power Measurement ACME Board**. It contains high Accuracy sensors for current, voltage and power. This device sense the power supply to the monitored device. The developed plugin collects from the measuring device throw an Ethernet connection, configures the measuring device, and collects the required data.

- **Management of monitoring configurations on heterogeneous systems.** The PHANTOM Monitoring Framework allows an administrator to register the description of the hardware components of the compute nodes, as well as their default configuration for monitoring. This task would normally be carried out only when new nodes are added to the system or when the existing nodes are updated. In

this way, the use of the framework is eased to the end users, since it frees the users of the configuration task as well as the necessary expertise to do it.

- **Data representation schema for the Monitoring Database:** The structure of the data registered is faithful representation of the each evaluation of each application. It contains the devices with its characteristics, and the structure of the workflows. The later contains the monitored measurements and the reference of the hardware were ran each of the different components of each application.

Next, are listed some other innovations on the particular components of the PHANTOM Monitoring Framework:

MF-Server

- The monitoring server improves the efficiency by allowing the data analytics to be performed where the data is stored. It is more efficient than sending a large amount of data to be processed in another location.

- The separation of gathering, storing and analysis technologies. The MF-Server, playing the role of the interface, can provide a consistency filter of the collected data, and filter possible errors. The analysis of previous executions and generation of statistics is done by the Execution Manager which can prepare the request of such results as soon as the metrics are available, improving the response time.

- The analysis functionality can be easily extended to the users' needs. The MF-Server is open-source and both support user-defined micro-queries directly submitted by the users.

- The users can access all the stored data in the database, which can support to debug their tools and analyse the behaviour of the user applications.

- Integration of the application-specific monitoring data into visualization back-ends. Tools like Grafana[1] and other open-source visualization solutions can easily be used.

Highly scalable on distributed platforms: MF-server as well as the ElasticSearch database, can run on distributed platforms.

MF-Library

The following major innovations are identified for the Monitoring Library:

- **Highly customizable monitoring settings.** The PHANTOM Monitoring Framework was designed with the user-friendliness in mind – the users can specify the metrics to be automatically collected, the flush time interval, the configuration of

[1] https://grafana.com/

collected metrics, etc., independently for each application. These customization options are absent from alternative approaches.

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

- **High-accuracy time measurements.** Unlike the conventional solutions that support the range of measurement in the order of seconds, the MF-Library allows monitoring metrics gathering in the range of microseconds.

  This accuracy is achieved by executing few machine instructions in the CPU to access to the hardware counters. The machine instructions which require each one few ns, thus only executing few of them will guarantee in the worst case the accuracy of few µs on GHz CPUs, and tens of µs on hundreds of MHz CPUs.

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

  Each user metric is composed of a text label and a value or a set of values, which are automatically timestamped. These metrics support the measurement of the work done, (for instance such hits can be the number of frames processed from a video sequence), which in conjunction with the energy measurements will allow proposing more efficient allocation of tasks. The user-defined metrics are also missing from existing tools.

3.1.5 Security

Including the refinement of the NGAC functional architecture to place the resource access path into the hands of the application developer, a lightweight NGAC Policy Server with RESTful Policy Query Interface API, and a declarative policy language accepted by the NGAC policy tool. This all makes access control a first-class part of the development model and automatically enforceable by the platform. An approach to providing “process isolation” to processes implemented as FPGA IP cores, and the restricted use of GPUs, along with the implementation and enforcement of the component network architecture through process isolation and communication constraints completes the PHANTOM platform as a heterogeneous platform in the MILS paradigm.

3.1.6 Model-Based Testing

MBT in PHANTOM enables both early validation and thorough test execution. The innovation of MBT in PHANTOM consists of the following points:

- **MBT in embedded environments**. MBT for embedded is challenging due to the heterogeneity and connectivity of embedded systems [5]. To support the MBT in embedded systems, system adapters have been designed and developed in individual use case embedded environments, as well as the PHANTOM integrated environment. As far as relevant system adapters are available, MBT is able to execute other test cases reusing the system adapters over the supported embedded environments.

- **MBT for component-based applications**. Taking advantage of the component-based feature of PHANTOM applications, we have designed model validation and performance estimation activities so analyze functional flow and non-functional dependency among components as to enable early functional and non-functional testing without executing the application.

- **MBT in parallel environments**. To fully take PHANTOM's parallel features into account, we have developed MBT models based on communicative state machine to capture both dynamic behaviors of components and the communications within an application, and thus the test case generated from the model are adapted to the parallel environment as well.

Additionally, based on the design and implementation of MBT in PHANTOM, EGM has submitted an application of innovation patent, and the application number of **MBT patent** is 1800821.

3.1.7 Resource, Application, Execution Managers

The PHANTOM Managers are supports on the coordination and integration of the different PHANTOM tools, applications and users.

The Resource, Application and Execution managers are lightweight services, with web and restful interfaces, which registers and support the progress of the PHANTOM toolset. In this section it is described their final design, architecture and their functionalities.

The major innovations are identified for the Managers are:

- **Flexibility.** The Resource Manager allows supporting heterogeneous infrastructure unlike other solutions. In case of the other managers, there were not available alternative solutions that can be customized as the needed communications between tools.

- **Service of notification of updates.** The Managers provides a subscription mechanism on the updates of their respective registered data (Execution of apps/ preparation stage of Apps/ status of the resources). The notifications consists on sending the JSON data of the respective updated data. This is useful in scenarios such as PHANTOM users trigger on a particular event, like the availability of files for a particular tool, the availability of a particular hardware, the request of an execution of an application, or the end of a particular execution.

3.2 FULFILLING THE PHANTOM AMBITIONS

3.2.1 Parallelization Toolset

The Parallelization Toolset's main ambitions concentrate on the exploitation and extension of existing parallelization tools to transform individual software components into target specific components. The developed toolset includes the production of parallelized versions that can be accelerated when offloaded on SMPs, GPU compatible versions that given the right kernel implementations can offload data on GPUs for accelerated computations and transformed versions that can use reconfigurable FPGA devices to efficiently implement mathematical functions.

Exploitation of SMP architectures
Through the adoption of the autoPar tool and its adaptation to real-world software, the PT is able to offer real solutions for serial code to exploit highly resourced architectures that offer multiple general-purpose processing cores to accelerate execution.

IP Core Marketplace and IP Core Generator
The main advantage of the IP Core Marketplace and IP Core Generator is to provide a way for the user to exploit the benefits of FPGA hardware without the need for any extra effort or knowledge about reconfigurable hardware. The Marketplace provides optimized IP cores of common mathematical algorithms that can be reused in many different applications, while the IP Core Generator allows the user to run part of their specific application in FPGA hardware. The main innovations include:

- Collection of curated IP cores for common mathematical algorithms optimized by hardware engineers that are easy to integrate in any application.
- Ability to create IP cores and the software-hardware adapter from normal C/C++ code without developer intervention.
- Allows a software developer to exploit hardware resources without need for reconfigurable hardware knowledge.
- Simple and straightforward integration with the original application.

Exploitation of the Programming Model
The component-based nature of the Programming Model allows the extension of the aforementioned support into fine-grained deployments that can combine all the above ambitions into component-based parallelization according to the characteristics of each software unit. So, the functionality of the PT tool flow is extended with the support of GPU-based or FPGA-based components resulting in a full-platform support, unlocking multiple parallelization capabilities. Deployment on multiple devices can be achieved, hence coverage for variable-type components with different requirements is guaranteed.

3.2.2 Multi-Objective Mapper (MOM)

The main contribution of MOM is the optimization against multiple objectives at functional and non-functional level, where target requirements include power, execution time, reliability, memory utilization, data movement and security.

MOM is based on the use of multidimensional, adaptive and evolutionary optimization in order to generate deployment plans that address the idea of multi-dimensional optimization in terms of optimizing against multiple objectives and targeting systems with increased heterogeneity, thus targeting distributed and massively parallel computing, in full consistency with the PHANTOM Component-based Programming Model. The exact objectives to optimize are actually derived by the analysis of PHANTOM use cases and the capabilities disclosed by the other PHANTOM technologies during the project execution. MOM's design supports by definition all 3 types of HW platforms (CPU, GPU, FPGA). For the PHANTOM prototype, all use cases have demonstrated the capability of MOM to select the optimal mapping from all three supported platforms mentioned above.

3.2.3 Deployment Manager

The development of an additional tool to coordinate the compilation and execution procedures of the parallel application components was considered necessary to reduce the complexity that arises when using heterogeneous platforms. Results from the preceding tools are used for the configuration of the application's deployment which is automated, creating in this way another layer of abstraction for the user to facilitate the procedure and complete the toolflow's platform-agnostic nature.
The Deployment Manager aims to integrate the application deployment procedure into the whole toolflow of PHANTOM, abstracting from the user the low-level instrumentation that is needed for the application's execution on a heterogeneous infrastructure. The tool:

- Supports all 3 types of hardware platforms (CPUs, GPUs and FPGAs).
- Implements code refactoring to integrate the parallelized components into a ready to execute application in a platform-agnostic manner towards the user
- Orchestrates the communication of the application components using the PHANTOM communication protocols (Queue, Shared, Signal) built upon standard communication technologies (MPI, POSIX) and taking into account memory locality optimizations
- Performs the compilation procedure, in an efficient manner, avoiding unnecessary compilations or data transactions with the PHANTOM Repository
- Performs the execution through MPI runtime environment, monitoring the procedure, while keeping the user informed through logs about any unexpected behavior, thus enabling the user to fix/improve the application design before redeploying it

3.2.4 Monitoring Framework

The objective of the PHANTOM Monitoring Framework is to enable the application optimization based on the understanding of both software non-functional properties and hardware quality attributes with regards to performance and energy consumption.
In order to achieve the requirements of the project, we extended the monitoring framework developed in the ATOM, Excess and DreamCloud eu-projects. In the Phantom project it was added monitoring support at the application level. For the purpose it has been developed monitoring libraries which provide monitoring support to the applications to be monitored.

In addition, support was developed for monitoring heterogeneous hardware (including Nvidia-GPUs and Xilinx FPGAs) in the monitoring libraries, as well as additional plugins for the monitoring client at the system level.

3.2.5 Security

The PHANTOM platform and tools support component network execution integrity down to the level of the individual processing elements, including FPGAs and GPUs. Though existing MILS platforms and tools were appropriately found to not be the best choice for deployment of PHANTOM, due to their scarcity and costliness, the MILS principles were applied to the design and implementation of the PHANTOM platform to achieve execution integrity. The PHANTOM access control system has been designed in compliance with the NGAC standard and integrated with the PHANTOM Repository and other PHANTOM managers. The PHANTOM monitoring platform has been developed to meet also security auditing requirements as well as its primary function of gathering application runtime metrics.

3.2.6 Model-Based Testing

Through four MBT activities (i.e. model validation, performance estimation, functional testing and non-functional testing) in two phases (i.e., early validation and test execution), MBT in PHANTOM provides an end to end and enhanced Model-Based Testing system for both functional and non-functional verification and validation of PHNATOM applications over several heterogeneous technologies and computing continuum devices brought by PHANTOM platform. Model simulation and test generation are integrated, while a number of MBT components are developed to implement and achieve all MBT activities. Existing MBT modelling methods that rely on the UML are extended to a communicating state machine meta model for the MBT model creation in PHANTOM in order to consider the internal communication of application components and support simulation and generation. Test cases generated by tools are concretized by TTCN-3 publisher and Codedcs/Decodes for automated test execution. Test execution does not only provide functional and non-functional insights for developers, but also assists PHANTOM module MOM to collect performance metrics of different component for mapping references. Furthermore, users and developers can easily evaluate their applications by their own to develop MBT models following the specific modelling language and conduct MBT activities by use of the entire testing system. Based on the evaluation at the second year of the project, the partners are satisfied with the testing efficiency, complexity and testing coverage, while further evaluation will be made by the end of the project.

One particularly important aspect that MBT work has achieved its ambition is that through two early validation stage activities, the testing system enables developers to test very early in the life cycle system their applications in parallel of the development. The MBT activities are able to eliminate human error in an early stage and prevent them from escalating to implementation level. An innovation patent is submitted based on the MBT work in PHANTOM, while a commercial offer of the entire testing systems has been made and presented in French Industry event.

3.2.7 Application, Execution, and Resource Managers

These services support interaction between the different PHANTOM tools as well as to keep track of the evolution of the applications, executions, and hardware devices. There are no existing tools that allow the personalization of the stored information nor the support for the interaction among the PHANTOM tools.

4. CONCLUSION

This document has described an overview of the entire PHANTOM toolflow and platform, and the ways in which it supports developers to build complex, distributed systems. Unlike other approaches, PHANTOM supports a range of platforms from small embedded devices up to high performance computing infrastructures. PHANTOM defines a component-based programming model in which components explicitly define the shared data items they use. Components can also define non-functional requirements (power, execution time, memory allocation, platform reliability), security requirements (isolation and integrity of communications), and deployment restrictions. Additionally, an iterative development cycle is supported to assist the developer create a deployment which meets the system's requirements.

- A Parallelisation Toolset creates parallelised versions of the system components that use SMPs, GPUs, or FPGAs.
- A Multi-Objective Mapper (MOM) automatically creates optimised deployments in an attempt to meet non-functional requirements (such as minimising power use or execution time).
- A Secure Execution Platform implements the desired security features of the developer's application.
- The development process is supported by a Model-Based Testing framework, which uses runtime data (component power use, execution time, user-defined metrics) collected by an automatic Monitoring Framework.
- PHANTOM's Deployment Manager uses code generation to automatically implement the deployment choices made by the MOM, and to automatically invoke the monitoring framework, allowing for fast and error-free design space exploration. Automated deployment takes place at the end of each iteration, taking into account the availability and status of the deployment platform, information provided at real-time by the Resource Manager.
- The whole process is being coordinated by the Application Manager that keeps track of the tools' status and allows the communication between them. Additionally, the Execution Manager stores information about each individual execution which becomes available to the other tools.

This document represents the final design for the PHANTOM framework including the latest refinements of the platform and programming model identified during the development process.

5. REFERENCES

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

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

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

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

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

[6] INCITS 499-2013, Information technology – Next Generation Access Control – Functional Architecture, InterNational Committee for Information Technology Standards, Cyber Security technical committee 1, 2013.

6. APPENDIX 1. EXAMPLES OF NGAC SECURITY POLICIES AND TOOL RUNS

A graphical representation of two attribute-based policies are shown in Figure 26. Policy (a) defines the Project Access policy class that aggregates users by groups and objects by projects. Access permissions to project objects are expressed by groups relative to projects. Policy (b) defines a File Management policy class that illustrates how access to personal files may be restricted or shared.

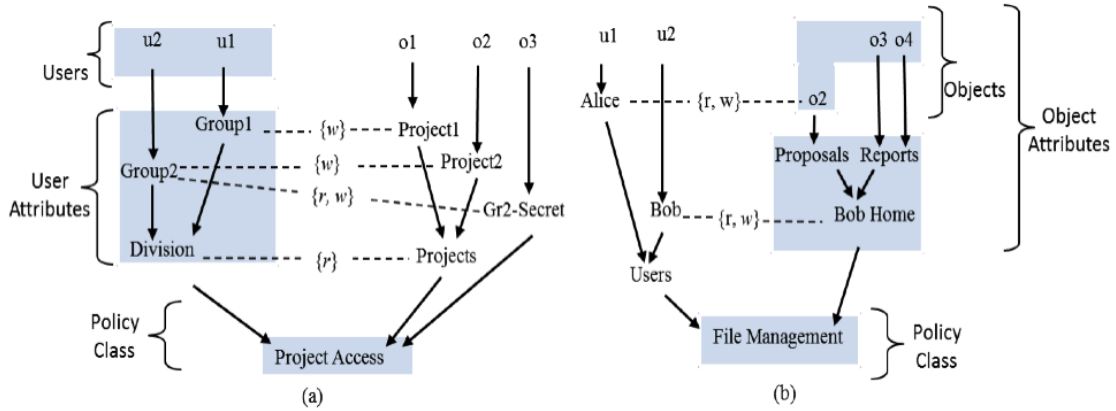


Figure 26: Two attribute-based security policies

Assignment relations are represented as arrows. Association relations are represented by a dashed line and include the set of access rights shown on the line. The derived privileges (allowed user-operation-object tuples) of each of the policies separately are shown in Figure 27.

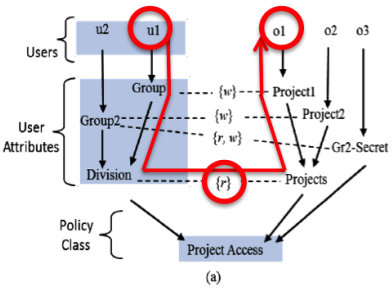
(u1,r,o1),(u1,w,o1),(u1,r,o2),(u2,r,o1), (u2,r,o2),(u2,w,o2),(u2,r,o3),(u2,w,o3)	(u1,r,o2),(u1,w,o2),(u2,r,o2),(u2,w,o2), (u2,r,o3),(u2,w,o3),(u2,r,o4),(w2,w,o4)

Figure 27: Derived privileges of policies (a) and (b)

A run of the ngac policy tool is shown in Figure 28. The first column shows the declarative policy language specification of Policy (a). The next three columns show test of some internal functions, including the logical object space visible to the different users. The last column shows the access control tests, which are answered by `permit` or `deny`. The `permit` response by ngac to the first query, `access('Policy (a)',(u1,r,o1))`, is highlighted. This response corresponds to the path highlighted in the graphical representation of the policy, specifically, user `u1` belongs to user attribute `Group1` which in turn belongs to `Division`, which is granted `r` access to object attribute `Projects`, which contains `Project1`, which in turn contains object `o1`.

The PM server receives direction from the PM admin tool in the form of imperative commands that build or modify the security policy database. The ngac policy tool can also be able to convert its declarative policy description to the imperative command form used by the PM server reference implementation. This provides the ability to develop a policy using the ngac policy tool and to deploy it in the PM server.

```
ngac> proc(testA,verbose).
> newpol('Policy (a)').
> los('Policy (a)').
user(u1)
user(u2)
object(o1)
object(o2)
object(o3)
user_attribute(Division)
user_attribute(Group1)
user_attribute(Group2)
object_attribute(Gr2-Secret)
object_attribute(Project1)
object_attribute(Project2)
object_attribute(Projects)
policy_class(Project Access)
assign(u1,Group1)
assign(u2,Group2)
assign(Group1,Division)
assign(Group2,Division)
assign(o1,Project1)
assign(o2,Project2)
assign(o3,Gr2-Secret)
assign(Project1,Projects)
assign(Project2,Projects)
assign(Division,Project Access)
assign(Projects,Project Access)
assign(Gr2-Secret,Project Access)
associate(Group1,[w],Project1)
associate(Group2,[w],Project2)
associate(Group2,[r,w],Gr2-Secret)
associate(Division,[r],Projects)
Connector(PM)

> dps('Policy (a)').
Group1,r,Project1
Group1,r,Project2
Group1,r,o1
Group1,r,o2
Group2,r,Project1
Group2,r,Project2
Group2,r,o1
Group2,r,o2
u1,r,Project1
u1,r,Project2
u1,r,o1
u1,r,o2
u1,w,o1
u2,r,Project1
u2,r,Project2
u2,r,o1
u2,r,o2
u2,r,o3
u2,w,o2
u2,w,o3

> aoa(u1).
Project1
Project2
Projects
o1
o2

> minaoa(u1).
Projects

> userlos('Policy (a)',u1).
V= PM
 Project Access
 o1
 o2
 Project1
 Project2
 Projects
E= Project Access,PM
 Projects,Project Access
 o1,Project1
 o2,Project2
 Project1,Projects
 Project2,Projects

> aoa(u2).
Gr2-Secret
Project1
Project2
Projects
o1
o2
o3

> minaoa(u2).
Gr2-Secret
Projects

> userlos('Policy (a)',u2).
V= PM
 Project Access
 o1
 o2
 o3
 Project1
 Project2
 Gr2-Secret
 Projects
E= Project Access,PM
 Gr2-Secret,Project Access
 Projects,Project Access
 o1,Project1
 o2,Project2
 o3,Gr2-Secret
 Project1,Projects
 Project2,Projects
```




```
ngac> proc(queryA,verbose).
> access('Policy (a)', (u1,r,o1)).
permit
> access('Policy (a)', (u1,w,o1)).
permit
> access('Policy (a)', (u1,r,o2)).
permit
> access('Policy (a)', (u1,w,o2)).
deny
> access('Policy (a)', (u1,r,o3)).
deny
> access('Policy (a)', (u1,w,o3)).
deny
> access('Policy (a)', (u2,r,o1)).
permit
> access('Policy (a)', (u2,w,o1)).
deny
> access('Policy (a)', (u2,r,o2)).
permit
> access('Policy (a)', (u2,w,o2)).
permit
> access('Policy (a)', (u2,r,o3)).
permit
> access('Policy (a)', (u2,w,o3)).
permit
ngac>
```

Figure 28: Description of Policy (a) and interactive test

An example policy representing the GMV use case expressed in the declarative language is presented in Figure 29 and the graphical representation of the policy is shown in Figure 31. A run of the policy tool to test the policy is shown in Figure 30.

```
policy('review', 'Surveillance Database', [

      % list of users
      user('satellite'), % producer of the image and related info
      user('geo_entity'), % producer of earth geographic info
      user('simons_app'), % program that will access the available data
      user('intruder'), % menace
      user_attribute('Producer'), % able to read and write any file
      user_attribute('Geo_Producer'), % able to read/write any geographic file
      user_attribute('Consumer'), % able to read any file
      user_attribute('Users'),

      % list of objects
      object('metadata'),
      object('image_data'),
      object('coastline'),
      object_attribute('Image'),
      object_attribute('Aux file'),
      object_attribute('Data'),

      policy_class('Surveillance Database'),
      connector('pm'),

      % list of relations of users
```

```
      assign('satellite', 'Producer'),
      assign('geo_entity', 'Geo_Producer'),
      assign('simons_app', 'Consumer'),
      assign('satellite', 'Users'),
      assign('simons app', 'Users'),
      assign('geo_entity', 'Users'),

      % list of relations of objects
      assign('metadata', 'Image'),
      assign('image data', 'Image'),
      assign('coastline', 'Aux file'),
      assign('Image', 'Data'),
      assign('Aux file', 'Data'),

      % other relations
      assign('Users', 'Surveillance Database'),
      assign('Data', 'Surveillance Database'),

      % access policies
      associate('Producer', [read, write], 'Data'),
      associate('Geo_Producer', [read, write], 'Aux file'),
      associate('Consumer', [read], 'Data')
]).
```

Figure 29: Definition of the policy for GMV use case

```
import(policy('simons_ngac_policy.pl')).
newpol('review').

access('review', (satellite, read, metadata)). % permit
access('review', (satellite, write, metadata)). % permit
access('review', (satellite, read, image_data)). % permit
access('review', (satellite, write, image_data)). % permit
access('review', (satellite, read, coastline)). % permit
access('review', (satellite, write, coastline)). % permit

access('review', (simons_app, read, metadata)). % permit
access('review', (simons_app, write, metadata)). % deny
access('review', (simons_app, read, metadata)). % permit
access('review', (simons_app, write, metadata)). % deny
access('review', (simons_app, read, coastline)). % permit
access('review', (simons_app, write, coastline)). % deny

access('review', (geo_entity, read, metadata)). % deny
access('review', (geo_entity, write, metadata)). % deny
access('review', (geo_entity, read, image_data)). % deny
access('review', (geo_entity, write, image_data)). % deny
access('review', (geo_entity, read, coastline)). % permit
access('review', (geo_entity, write, coastline)). % permit

access('review', (intruder, read, metadata)). % deny
access('review', (intruder, write, metadata)). % deny
access('review', (intruder, read, image_data)). % deny
access('review', (intruder, write, image_data)). % deny
access('review', (intruder, read, coastline)). % deny
access('review', (intruder, write, coastline)). % deny
```

Figure 30: Test of the GMV policy in the policy tool

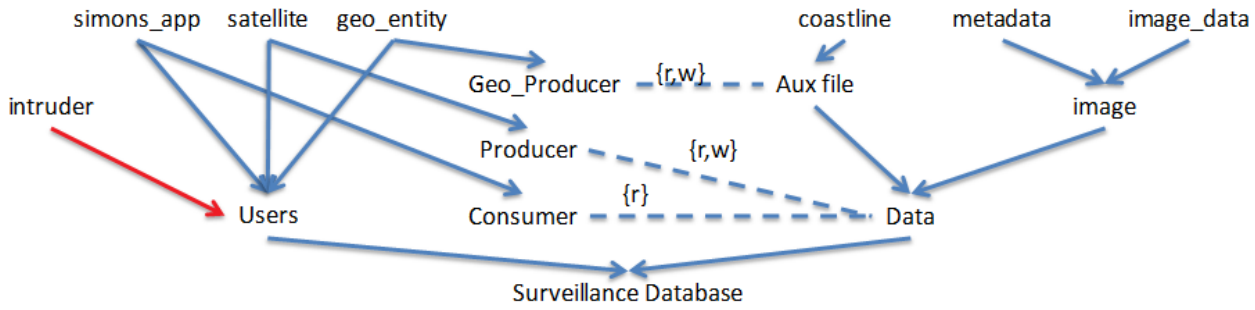


Figure 31: Schema of policy from GMV use case