PHANTOM

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

Project Number 688146

D5.5 – Final developers' documentation

Version 1.2
17 May 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
0.8	First release for internal review	27/02/2019
0.9	Updates reflecting integrations and use case feedback	28/03/2019
1.0	Further partner updates	01/04/2019
1.1	Final Release	02/04/2019
1.2	Final Release with EC required updates	17/05/2019

TABLE OF CONTENTS

INDEX OF FIGURES

EXECUTIVE SUMMARY

This document provides the final documentation for the PHANTOM toolchain. It includes guidelines that need to be followed when placing an application for deployment in the PHANTOM framework, so as to facilitate the use of the PHANTOM tools. In section 2, a basic deployment of the tools is described for a "Hello World" application to facilitate the familiarity of the user with the toolflow using the Integrated Reference System Virtual Machine. In section 3, the characteristics of the Programming Model that can be adopted by the user are described, while the deployment guides for each PHANTOM tool are presented in section 4, in terms of guidelines about the individual deployment of each tool and the whole toolchain with details that will ensure their installation and execution on any platform that satisfies the dependencies specified. The deployment guides concern all the tools of the PHANTOM toolchain, namely Repository, Parallelization Toolset, IP Core Generator, IP Core Marketplace, Multi-Objective Mapper (Generic and Offline), Deployment Manager, PHANTOM FPGA Linux Infrastructure, Monitoring Framework Server and Client, Security Manager, Model-Based Testing, Application Manager, Resource Manager, Execution Manager, as well as the Integrated Reference System user scripts. The usage capabilities that are provided by PHANTOM are described in the following sections with specific guidelines. For details about each individual tool's functionality see D1.4, D2.2, D3.2 and D4.4.

1. INTRODUCTION

The design details of the different PHANTOM technologies have been thoroughly described in previous deliverables (see D1.4, D2.2, D3.2, D4.4). The deployment of the tools has also been presented in the context of the Integrated Reference System in D5.2. In this document, details are given concerning the individual deployment of the different tools, as well as the initial set-up and deployment of a Hello World application using PHANTOM.

2. GETTING STARTED

In this section, the deployment of a simple application exchanging data is described to showcase the steps that should be followed to develop a PHANTOM application. These steps include the following:

1. Defining the Application Model
2. Defining the System Model
3. Structuring the Application and System models in XML format
4. Creating application components (C,C++ code)
5. Using the Programming Interface and code annotations
6. Placing application files (source, input, description files) on the Repository
7. Initiating the PHANTOM tools

This tutorial is designed to work with the help of the Integrated Reference System Virtual Machine (see D5.2 for details) which automates a lot of the work needed by the developer. The same procedure can be followed for the case of individual deployment of the tools which will also require the manual deployment of the tools as described in section 4.

2.1 CREATING A PHANTOM APPLICATION

2.1.1 Defining the Application Model

The first thing a PHANTOM developer needs to do is define the different application components that are going to comprise the whole application and the communication objects that are going to be used.

For this tutorial, a simple application with 3 components is used. To showcase the functionality offered by the PHANTOM Programming Interface, the components will be exchanging data using the Shared and Queue protocols, while synchronizing using the Signal protocol. In specific, the following transactions are observed in Figure 1.

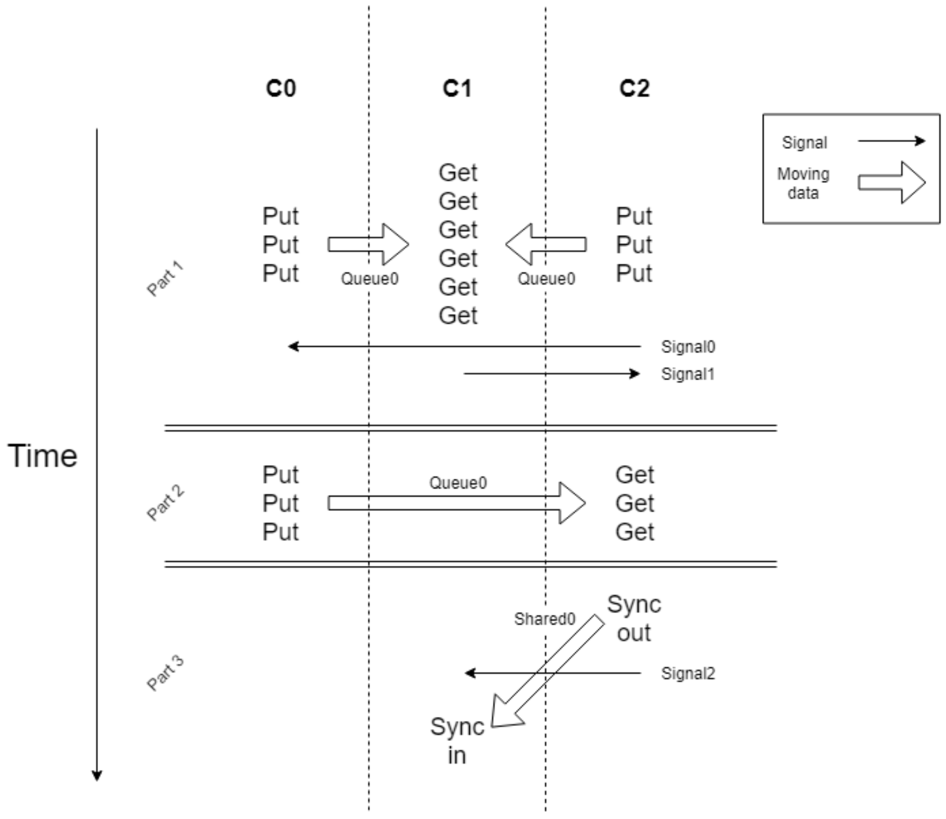


Figure 1: Example application data flow

In the first part the application, arrays of integers are inserted in the queue that is used between all three components by components C0 and C2 and are collected by component C1. Signals and the blocking attribute of the *phantom_queue_get()* function are used for the coordination of the data transfers that guarantees the deterministic behavior of the application.
In the second part, arrays of integers are also moved from component C0 to C2. Here, coordination is guaranteed by the blocking queue.
In the third part, a shared array is used between components C1 and C2 and consistency is guaranteed with the use of a signal.

The source code of the components defines the behavior of each component in separate, but in order to define the application as a whole a high-level description is required by the developer for the definition of the application's extrovert behavior. For our example, the components are modeled as shown in Figure 2 using XML format.

```
<component name="C0" type="asynchronous">
    <source file="C0.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" GPU="no" FPGA="no"/>
</component>

<component name="C1" type="asynchronous">
    <source file="C1.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" GPU="no" FPGA="no"/>
</component>

<component name="C2" type="asynchronous">
```

```
    <source file="C2.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" GPU="no" FPGA="no"/>
</component>
```

Figure 2: Example application's components definition

In Figure 3 the definition of the corresponding communication objects are also shown.

```
<comm-object   name="Queue0"   object-class="FIFO"   item-size="1"   size="100"
type="Queue">
       <source name="C0" port-name="inQueue00" type="in"/>
       <target name="C1" port-name="outQueue01" type="out"/>
       <target name="C2" port-name="outQueue02" type="out"/>
</comm-object>

<comm-object   name="Shared0"   object-class="shared"   item-size="4"   size="10"
type="Shared">
       <source name="C2" port-name="inShared0" type="in"/>
       <target name="C1" port-name="outShared0" type="out"/>
</comm-object>

<comm-object  name="Signal0"  object-class="signal"  type="Signal"  item-size="1"
size="1">
       <source name="C0" port-name="inSignal0" type="in"/>
       <target name="C2" port-name="outSignal0" type="out"/>
</comm-object>

<comm-object  name="Signal1"  object-class="signal"  type="Signal"  item-size="1"
size="1">
       <source name="C1" port-name="inSignal1" type="in"/>
       <target name="C2" port-name="outSignal1" type="out"/>
</comm-object>

<comm-object  name="Signal2"  object-class="signal"  type="Signal"  item-size="1"
size="1">
       <source name="C2" port-name="inSignal2" type="in"/>
       <target name="C1" port-name="outSignal2" type="out"/>
</comm-object>
```

Figure 3: Example application's communication objects definition

Both components and communication objects are defined in the Component Network XML file which is thoroughly discussed in other deliverables (D1.2, D1.3, D1.4).

In the following sections, this example will be used to guide the reader through the different steps.

2.1.2 Defining the System Model

Apart from the Application Model the System model is also defined by the user and contains all necessary information about the hardware components that are available (single-CPU nodes, SMPs, CPU-GPU systems, FPGAs). For our example the description file (Platform Description XML file – also described in detail in D1.2, D1.3, D1.4) is shown in .

```
<platform name="localhost" xsi:noNamespaceSchemaLocation="./hw_phantom.xsd">
    <device name="DevelopmentMachine" type="CPU-SMP" reliability="5">
        <processing-node name="unit1" type="CPU-SMP" architecture="SMP">
            <processor name="local" type="INTEL-COREi7">
                <configuration name="core number" value="4"/>
                <configuration name="cpu frequency" value="2.6" unit="GHz"/>
                <configuration name="bytespercycle" value="1"/>
                <memory name="MEM1" type="RAM" size="8192" size-unit="MB"
access-time="1" access-time-unit="ns/word"/>
            </processor>
        </processing-node>

        <comm_interface name="enp0s3" type="Ethernet Network">
            <configuration name="speed" value="100" units="MBit/s"
ip="localhost" user="demo"/>
        </comm_interface>
    </device>
</platform>
```

Figure 4: Example application's Platform Description

2.1.3 Creating application components

Each application component is mapped to its own C/C++ source file which should contain at least one function with the same name as the source file. Additionally, the phantom library needs to be included in every component, so that the PHANTOM framework functions and structures can be exploited. For our example application, the definitions of the components are displayed in Figure 5.

```
// C0.cpp

#include "../phantom/phantom.h"

void C0() {
    …
}

------------------------------------------------------------------------

// C1.cpp

#include "../phantom/phantom.h"

void C1() {
    …
}

------------------------------------------------------------------------

// C2.cpp

#include "../phantom/phantom.h"

void C2() {
    …
}
```

Figure 5: Component function definitions

2.1.4 Using the Programming Interface and code annotations

In order to use the functionalities provided by the Programming Interface (also including the Monitoring Interface), the corresponding initializations should be made. In specific, every communication object should be declared using the corresponding high-level annotations, while a unique pointer also needs to be initialized for the use of the communication functions.

See Figure 6 for the example application's initializations.

```
// C0.cpp

#include "../phantom/phantom.h"

void C0() {
    phantom_monitor *monitor = phantom_monitor_init();

    . . .

    #pragma phantom queue inQueue00
    phantom_queue *C0_Queue0 = phantom_queue_init("inQueue00");

    #pragma phantom signal inSignal0
    phantom_signal *signal = phantom_signal_init("inSignal0");

    . . .
}

--------------------------------------------------------------------------------

// C1.cpp

#include "../phantom/phantom.h"

void C1() {
    phantom_monitor *monitor = phantom_monitor_init();

    . . .

    #pragma phantom queue outQueue01
    phantom_queue *C1_Queue0 = phantom_queue_init("outQueue01");

    phantom_shared *shared_object = phantom_shared_init("outShared0");

    #pragma phantom signal inSignal1
    phantom_signal *signal1 = phantom_signal_init("inSignal1");

    #pragma phantom signal outSignal2
    phantom_signal *signal2 = phantom_signal_init("outSignal2");

    . . .

    #pragma phantom shared outShared0
    int *shared_data1 = (int *)malloc(sizeof(int *) * 10);

    . . .
}

--------------------------------------------------------------------------------

// C2.cpp

#include "../phantom/phantom.h"

void C2() {
    phantom_monitor *monitor = phantom_monitor_init();

    . . .
    #pragma phantom queue outQueue02
    phantom_queue *C2_Queue0 = phantom_queue_init("outQueue02");

    phantom_shared *shared_object = phantom_shared_init("inShared0");

    #pragma phantom signal outSignal0
    phantom_signal *signal1 = phantom_signal_init("outSignal0");

    #pragma phantom signal outSignal1
    phantom_signal *signal2 = phantom_signal_init("outSignal1");

    #pragma phantom signal inSignal2
    phantom_signal *signal3 = phantom_signal_init("inSignal2");

    . . .

    #pragma phantom shared inShared0
    int *shared_data2 = (int *)malloc(sizeof(int *) * 10);

    . . .
}
```

Figure 6: Communication Object initializations

Finally, the functions provided by the Programming Interface can be used using the pointer variables declared in the previous step, like shown in Figure 7.

```
// C0.cpp

#include "../phantom/phantom.h"

void C0() {
    . . .

    for(i=0; i<3; i++) {
        phantom_queue_put(C0_Queue0,a[i]);
    }

    phantom_wait(signal);

    . . .

    for(i=0; i<3; i++) {
        phantom_queue_put(C0_Queue0,d[i]);
    }

    . . .

    phantom_mf_end(monitor);
}

--------------------------------------------------------------------------------

// C1.cpp

#include "../phantom/phantom.h"

void C1() {
    . . .

    for(i=0; i<6; i++) {
        a[i] = (uint8_t *)phantom_queue_get(C1_Queue0);
        . . .
    }
    phantom_notify(signal1);
    phantom_wait(signal2);
    phantom_synchronize(shared_object,shared_data1,0);

    . . .

    phantom_mf_end(monitor);
}

-------------------------------------------------------------------------------

// C2.cpp

#include "../phantom/phantom.h"

void C2() {
    . . .

    phantom_queue_put(C2_Queue0,a);
    phantom_queue_put(C2_Queue0,b);
    phantom_queue_put(C2_Queue0,c);

    phantom_notify(signal1);
    phantom_wait(signal2);

    for(i=0; i<3; i++) {
        d[i] = (uint8_t *)phantom_queue_get(C2_Queue0);
        . . .
    }

    . . .

    phantom_synchronize(shared_object,shared_data2,1);
    phantom_notify(signal3);

    . . .

    phantom_mf_end(monitor);
}
```

Figure 7: Example usage of Programming Interface

The complete components' source code for this example can be found in A.1.

2.2 USING THE INTEGRATED REFERENCE SYSTEM VIRTUAL MACHINE

Next, the recommended steps will be presented for correctly using the Virtual Machine, available at https://github.com/PHANTOM-Platform/Reference_VM:

1. **Update of the PHANTOM tools** – It is always recommended to run the script 'User-tools/management-scripts/update-tools.sh' to be sure that you are using the latest version of the PHANTOM tools. Please have in mind that if the update updates the 'update-tools.sh' script, then the 'update-tools.sh' script must be executed again in order apply the changes. To execute the update script run:

```
demo@ubuntu:~/phantom-tools/User-tools/management-scripts$ bash update-tools.sh
```

It is worth to notice that this script starts the local servers upon exit.

2. **Start local servers (optional)** - If the USER wants to use local instance of the servers and did not run the update script, then the servers must be started manually by running the command:

   ```
   demo@ubuntu:~/phantom-tools/User-tools/management-scripts$ bash start-servers.sh
   ```

   This is what we consider for this example.

3. **Copy of USER application to VM** - The next step consists in the deployment of the USER application files to be analysed inside the Virtual Machine at a location of the USER choice: `$CODE_DIR`.

   For this example, let's consider that we place the example's files in the directory:

   /home/demo/examples/tutorial

   Which should now have the following structure:

   ```
   turorial/
      Makefile
      cla.in
      src/
              C0.cpp
              C1.cpp
              C2.cpp
              example_lib.h
              phantom_user_defined_structs.h
      description/
              cpn.xml
              hw.xml
   ```

   Here we can spot a bunch of additional files which can be found at the corresponding location of the VM:

   - Makefile – this is the Makefile of the application.
   - cla.in – in this file the user can include input arguments for the application components as described in section 3.7. This is empty for our example since the components don't expect any input arguments.
   - example_lib.h – an included external library used by the components.
   - phantom_user_defined_structs.h – in this file the user needs to include any library that defines any structs that are used for shared objects. This is empty for our example since we don't use any user defined structures.

4. **Configuration of 'settings.py'** – The file 'settings.py' must be configured to according to the characteristics of the application and the intentions of the USER. The 'settings.py' is divide in 3 main zones:

   i. **Repositories configurations** – used for the USER specify the location of the repositories to be used (localhost or remote location) and credentials. For our example:

```
# Set the Repository IP address and port
repository_ip = "localhost"
repository_port = 8000
app_manager_ip = "localhost"
app_manager_port = 8500
exe_manager_ip = "localhost"
exe_manager_port = 8700
monitoring_ip = "localhost"
monitoring_port = 3033
resource_ip = "localhost"
resource_port = 8600

# Authentication credentials
user =
password =
```

   ii. **Application configurations** – Used for the USER to indentify application specific properties:

Name of the application to be used server and by PHANTOM tools to identify the application:

```
app_name = "tutorial"
```

Path for the root of the application's folder (to upload Makefile and cla.in)

```
root_path = "/home/demo/examples/tutorial"
```

Path for the folder with the source code

```
src_path = "/home/demo/examples/tutorial/src"
```

Path for the folder with description files (Component Network and Platform Description)

```
desc_path = "/home/demoexamples/tutorial/description"
```

Path for the folder with PHANTOM API files

```
phantom_path = "/home/demo//phantom-tools/PHANTOM_FILES"
```

Link to the where the marketplace is hosted and name of the folder where IPCores should be stored locally

```
ipMarket_path = "https://github.com/PHANTOM-Platform/PHANTOM-IP-Core-Marketplace.git"
```

ip_folder = "IPCore-MarketPlace"

Path for folder with application inputs
inputs_path = ""

Name of the component network file to be used
CompNetName = "cpn.xml"

Name of the platform description file to be used
PlatDesName = "hw.xml"

iii. **Tools Configurations** – This section contains the parameters for configuring the PHANTOM tools. In includes the path for each tool deployed on the machine. E.g.:

# MOM location
MOM_path = "/home/demo/phantom-tools/GenericMOM"

And tool specific arguments. E.g.:

PT_mode = "on" #operation mode: on | off - "on" to run PT normally, "off" to skip code analysis process

In this section can also be found the property for the address of the FPGA VM, as well as the SSH port to be used

FPGAVM_ip = ""
FPGAVM_port =

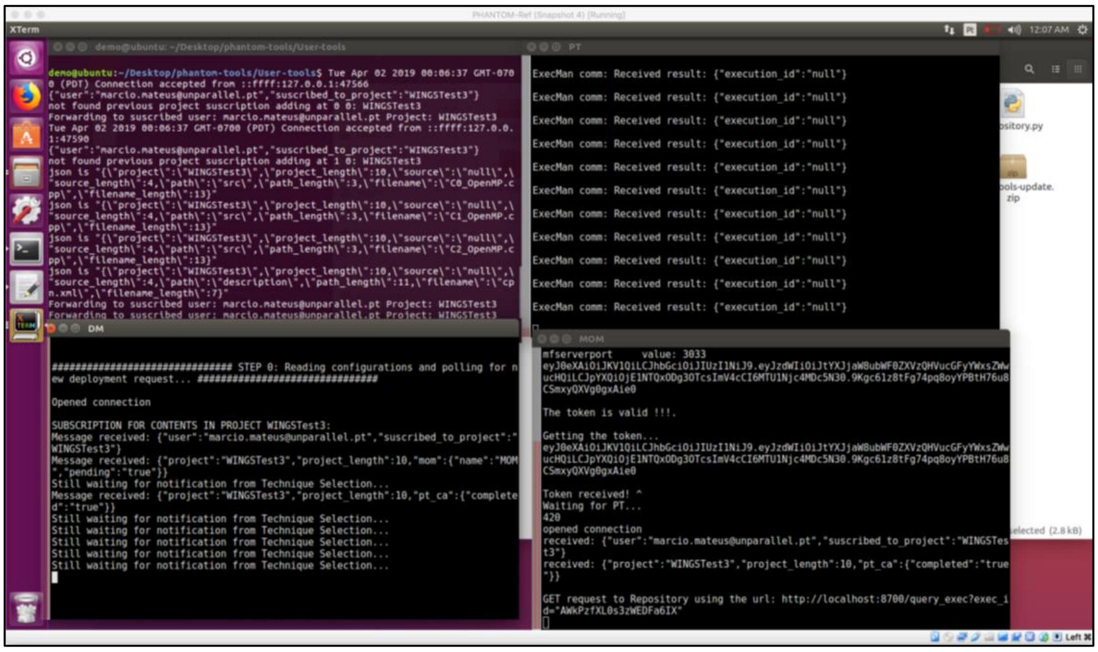
5. **Run the start script** – The last step consists in the execution of script that will: upload all the needed files to the specified repositories; register the application on the Application manager; and configure and launch each tool. To start this script run:

```
demo@ubuntu:~/phantom-tools/User-tools$./start-PHANTOM.py
```

This command as several options as shown the when using the '-h' flag:

```
demo@ubuntu:~/Desktop/phantom-tools/User-tools$ ./start-PHANTOM.py -h
usage: start-PHANTOM.py [-h] [-u] [-d] [-i] [-c] [-m] [-p]

Tool to support the execution of an application on PHANTOM Framework

optional arguments:
  -h, --help          show this help message and exit
  -u, --noUpload      Do not (re)upload the application to the repository.
                      (Application should be already in repository)
  -d, --onlyDesc      Only re-uploads the description files to the repository)
  -i, --skipInputs    Do not (re)upload the application inputs to the
                      repository. (Inputs should be already in repository)
  -c, --clean         Clean all the data in repositories and temporary cache
                      on PHANTOM tools. Automatically update PHANTOM_FILES
                      (-p)
  -m, --ipmarket      Uploads the IP Core Market place to the repository
  -p, --phantomfiles  Uploads the PHANTOM files (PHANTOM API and Monitoring
                      API)
```

During the execution of the application, the terminal windows are launched with the output of each tool. This way the USER can get a better understanding of what is happening and get feedback about any issue that may have occur with the execution of a tool. An example of an execution is shown in Figure 8, where the './startPHANTOM' script launched new terminals for the Parallelization Toolset (PT), Multi-Objective Mapper (MOM) and Deployment Manager (DM).



```
demo@ubuntu:~/Desktop/phantom-tools/User-tools$ Tue Apr 02 2019 00:06:37 GMT-070
0 (PDT) Connection accepted from ::ffff:127.0.0.1:47566
{"user":"marcio.mateus@unparallel.pt","suscribed_to_project":"WINGSTest3"}
not found previous project suscription adding at 0 0: WINGSTest3
Forwarding to suscribed user: marcio.mateus@unparallel.pt Project: WINGSTest3
Tue Apr 02 2019 00:06:37 GMT-0700 (PDT) Connection accepted from ::ffff:127.0.0.
1:47590
{"user":"marcio.mateus@unparallel.pt","suscribed_to_project":"WINGSTest3"}
not found previous project suscription adding at 1 0: WINGSTest3
json is "{\"project\":\"WINGSTest3\",\"project_length\":10,\"source\":\"null\",\
"source_length\":4,\"path\":\"src\",\"path_length\":3,\"filename\":\"C0_OpenMP.c
pp\",\"filename_length\":13}"
json is "{\"project\":\"WINGSTest3\",\"project_length\":10,\"source\":\"null\",\
"source_length\":4,\"path\":\"src\",\"path_length\":3,\"filename\":\"C1_OpenMP.c
pp\",\"filename_length\":13}"
json is "{\"project\":\"WINGSTest3\",\"project_length\":10,\"source\":\"null\",\
"source_length\":4,\"path\":\"src\",\"path_length\":3,\"filename\":\"C2_OpenMP.c
pp\",\"filename_length\":13}"
json is "{\"project\":\"WINGSTest3\",\"project_length\":10,\"source\":\"null\",\
"source_length\":4,\"path\":\"description\",\"path_length\":11,\"filename\":\"cp
n.xml\",\"filename_length\":7}"
Forwarding to suscribed user: marcio.mateus@unparallel.pt Project: WINGSTest3
Forwarding to suscribed user: marcio.mateus@unparallel.pt Project: WINGSTest3
```

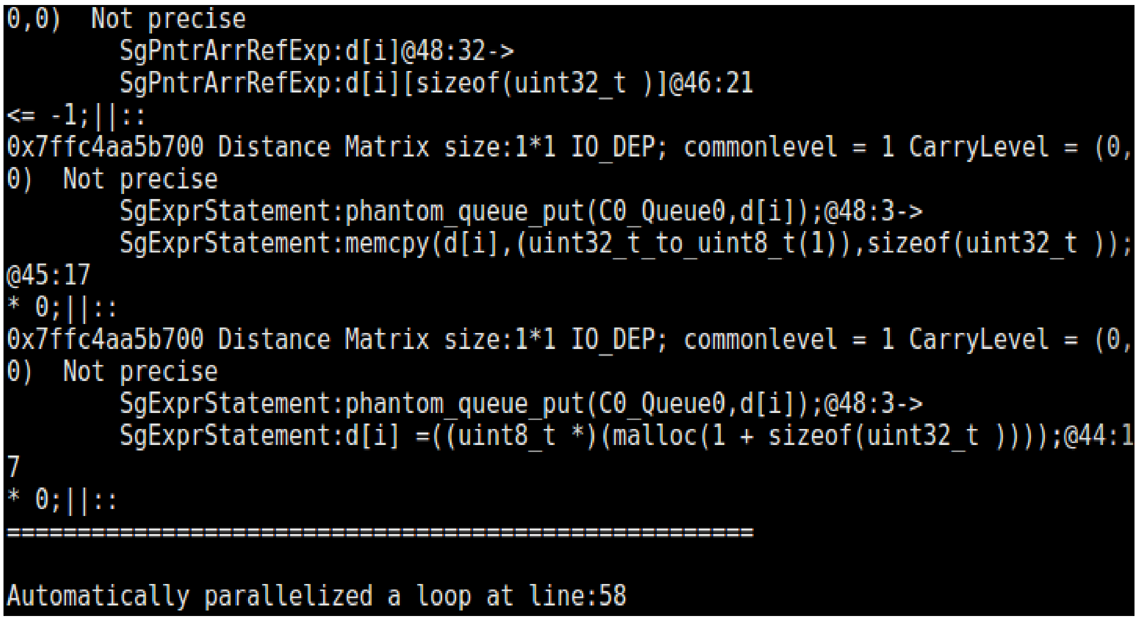
```
ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}

ExecMan comm: Received result: {"execution_id":"null"}
```

```
################################## STEP 0: Reading configurations and polling for n
ew deployment request... ##################################

Opened connection

SUBSCRIPTION FOR CONTENTS IN PROJECT WINGSTest3:
Message received: {"user":"marcio.mateus@unparallel.pt","suscribed_to_project":"
WINGSTest3"}
Message received: {"project":"WINGSTest3","project_length":10,"mom":{"name":"MOM
","pending":"true"}}
Still waiting for notification from Technique Selection...
Message received: {"project":"WINGSTest3","project_length":10,"pt_ca":{"complete
d":"true"}}
Still waiting for notification from Technique Selection...
Still waiting for notification from Technique Selection...
Still waiting for notification from Technique Selection...
Still waiting for notification from Technique Selection...
Still waiting for notification from Technique Selection...
```

```
mfserverport     value: 3033
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJtYXJjaW8ubWF0ZXVzQHVucGFyYWxsZWw
ucHQiLCJpYXQiOjE1NTQxODg3OTcsImV4cCI6MTU1Njc4MDc5N30.9Kgc61z8tFg74pq8oyYPBtH76u8
CSmxyQXVg0gxAie0

The token is valid !!!.

Getting the token...
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJtYXJjaW8ubWF0ZXVzQHVucGFyYWxsZWw
ucHQiLCJpYXQiOjE1NTQxODg3OTcsImV4cCI6MTU1Njc4MDc5N30.9Kgc61z8tFg74pq8oyYPBtH76u8
CSmxyQXVg0gxAie0

Token received! ^
Waiting for PT...
420
opened connection
received: {"user":"marcio.mateus@unparallel.pt","suscribed_to_project":"WINGSTes
t3"}
received: {"project":"WINGSTest3","project_length":10,"pt_ca":{"completed":"true
"}}

GET request to Repository using the url: http://localhost:8700/query_exec?exec_i
d="AWkPzfXL0s3zWEDFa6IX"
```

Figure 8 - Example of an execution of the start-PHANTOM.py script launching PT, MOM and DM

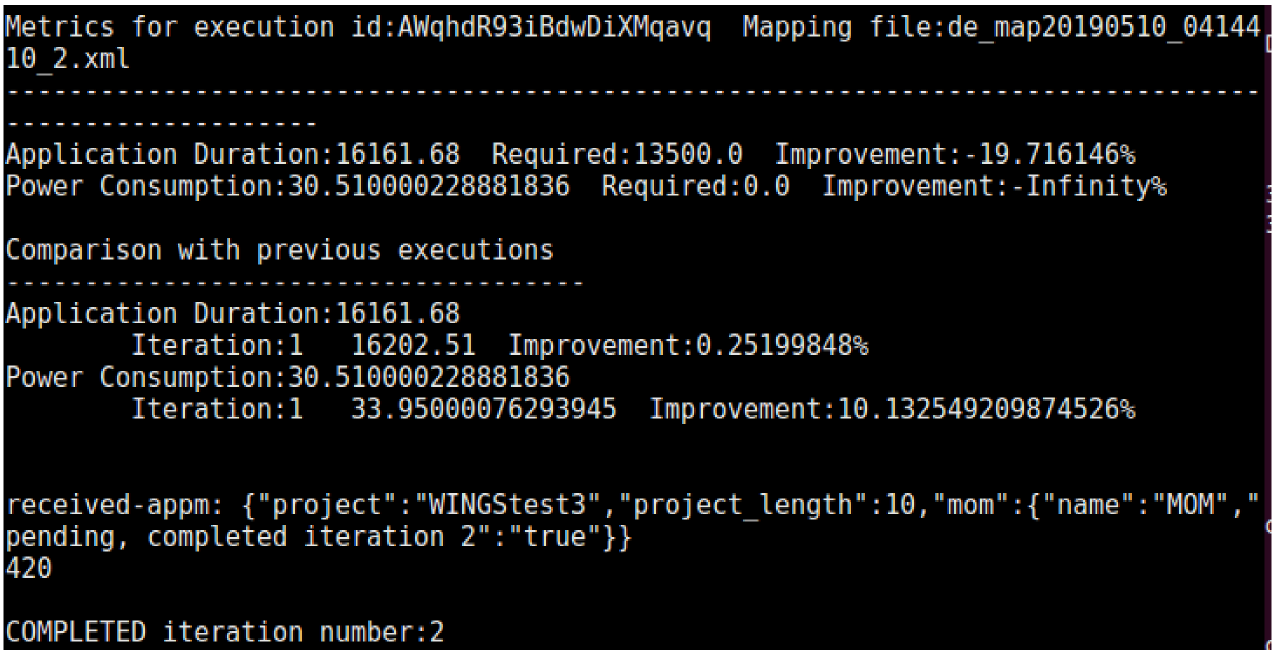
At the PT's terminal, output similar to the one appearing in Figure 9 demonstrates to the user the results of PT's Code Analysis pointing out the parallel regions that are identified and the dependencies for the non-parallel ones.



```
0,0)  Not precise
        SgPntrArrRefExp:d[i]@48:32->
        SgPntrArrRefExp:d[i][sizeof(uint32_t )]@46:21
<= -1;||::
0x7ffc4aa5b700 Distance Matrix size:1*1 IO_DEP; commonlevel = 1 CarryLevel = (0,
0)  Not precise
        SgExprStatement:phantom_queue_put(C0_Queue0,d[i]);@48:3->
        SgExprStatement:memcpy(d[i],(uint32_t_to_uint8_t(1)),sizeof(uint32_t ));
@45:17
* 0;||::
0x7ffc4aa5b700 Distance Matrix size:1*1 IO_DEP; commonlevel = 1 CarryLevel = (0,
0)  Not precise
        SgExprStatement:phantom_queue_put(C0_Queue0,d[i]);@48:3->
        SgExprStatement:d[i] =((uint8_t *)(malloc(1 + sizeof(uint32_t ))));@44:1
7
* 0;||::
=====================================================

Automatically parallelized a loop at line:58
```

Figure 9: Code Analysis results for component C0

At the MOM's terminal, the proposed mappings are displayed, as well as the metrics from the application's execution using them, as shown in Figure 10.



```
Metrics for execution id:AWqhdR93iBdwDiXMqavq  Mapping file:de_map20190510_04144
10_2.xml
--------------------------------------------------------------------------------
--------------------
Application Duration:16161.68  Required:13500.0  Improvement:-19.716146%
Power Consumption:30.510000228881836  Required:0.0  Improvement:-Infinity%

Comparison with previous executions
-------------------------------------
Application Duration:16161.68
        Iteration:1   16202.51  Improvement:0.25199848%
Power Consumption:30.510000228881836
        Iteration:1   33.95000076293945  Improvement:10.132549209874526%


received-appm: {"project":"WINGStest3","project_length":10,"mom":{"name":"MOM","
pending, completed iteration 2":"true"}}
420

COMPLETED iteration number:2
```

Figure 10: MOM's results after second iteration

The deployment procedure can also be monitored at the DM's terminal, while the output and error streams from the application's execution are stored in the corresponding files residing at the same directory as the tool's executable (for the VM this is: */home/demo/phantom-tools/DeploymentManager*).

3. PROGRAMMING MODEL

In this section, guidelines that need to be followed when placing an application for deployment in the PHANTOM framework are described to facilitate the use of the PHANTOM tools/servers.

Generic guidelines about the Programming Model have been described in previous deliverables. In this section, specific rules about the definition of the application components in the code are presented, as well as for the use of the PHANTOM Programming Interface and Monitoring Library.

3.1 APPLICATION MODEL

3.1.1 Components

A PHANTOM 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. They are initiated as separate and independent functions. These functions exist in the component's dedicated source file with the corresponding name. The developer is able to choose to pass command line arguments to the function. Finally, the return type of the functions should be *void**. So, the function's signature for a component with a source file *comp_example.cpp* should be like this:

```
void *comp_example(int32_t argc, char **argv)
```

or

```
void *comp_example()
```

Developers are free to include any external libraries they wish to use, as no compatibility issues were identified during development. On the other hand, in order for the components to use the PHANTOM libraries, the 'phantom.h' header file should be included, so that the necessary structures and functions are visible from the component main function.

3.1.2 Communication Objects

Components are defined as independent computational entities that don't interact directly. For this reason, an interface is provided by PHANTOM for the components to use to access data and coordinate with other components.

Four types of Communication Objects have been developed, Shared, Queue, Signal, Mutex (described in D1.2 and D1.4), 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`

3.2 Description Models

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 11.

```
<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 11: 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 12 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 12: Platform Description Example

Performance, power and security requirements can also be introduced by the user in the Component Network as shown in Figure 13.

```
<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 13: Requirement annotations in the Component Network

3.3 PROGRAMMING INTERFACE – MONITORING LIBRARY

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.

An important note here is that any communication object initializations should occur always locally inside the component's function. On the other hand, the object's pointer can be passed in any function the user requires to call the protocol functions (*phantom_queue_get*, *phantom_synchronize* etc.).

The user needs to also declare the corresponding communication object variables in order to use the protocol functions, which are cited in A.2.

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.

Same as with the communication object initializations, monitoring initializations should occur always locally inside the component's function, while the monitor's pointer can be passed in any function the user requires to call the monitoring functions, which are cited in A.2.

3.4 DATA TRANSFERS

The Queue Protocol has been developed to support object transfers of variable length. For that purpose, the objects are passed to the queue functions serialized including the size of the object in the 4 first bytes. Although the developers are free to develop their own functions for serialization, some prototype ones have been developed and are provided for the facilitation of the process. These are cited in A.2.

Concerning the Shared Protocol, the developer is able to transfer objects of user defined structures of static size. In order to enable such transactions, the user should include the libraries that define such structures in a dedicated header file for this purpose. This file is named '*phantom_user_defined_structs.h'* and it should be placed in the src directory.

3.5 FILE OPERATIONS

The PHANTOM Programming Model is armed 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. The functions are documented in detail in A.2.

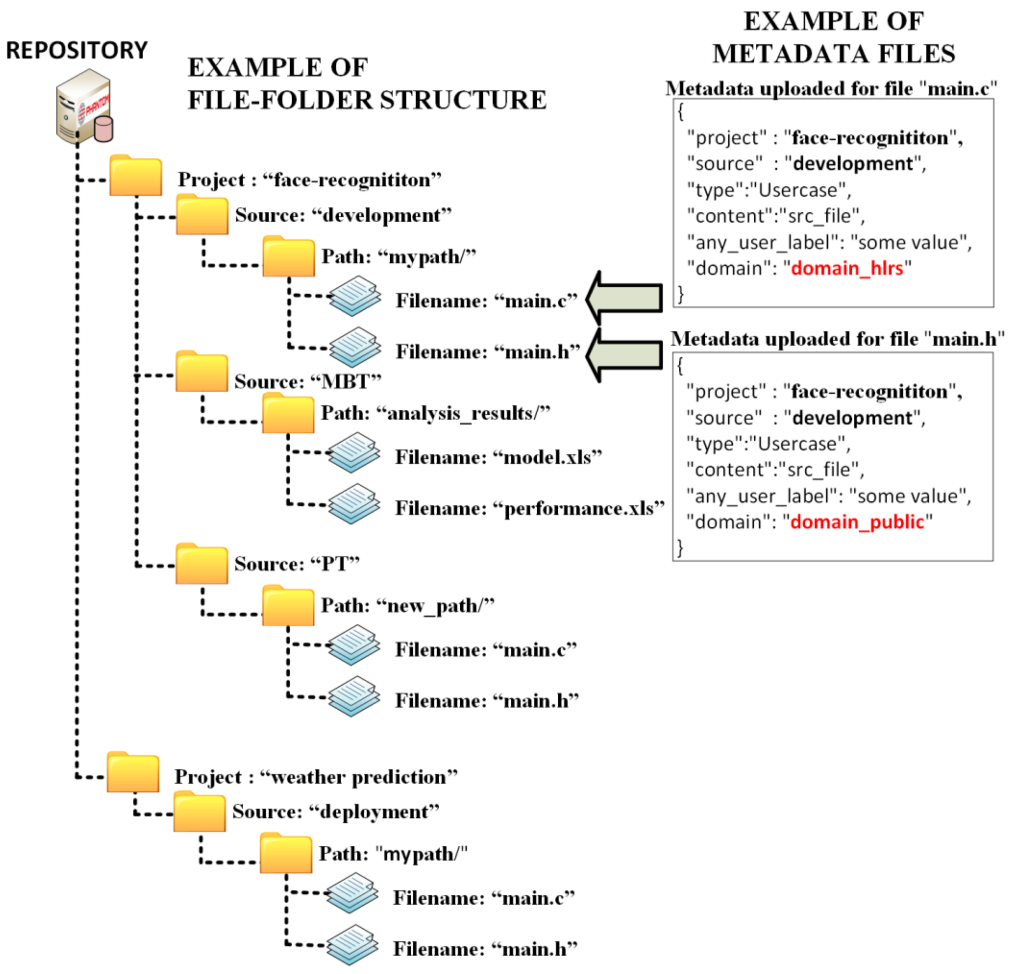
3.6 Optimization Annotations

The Programming Model includes assisting annotations for the static analysis of the code to enhance the parallelization results of the Parallelization Toolset (see D1.3). 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. They are also cited in A.2.

3.7 Placing developer's application in the PHANTOM Repository

The PHANTOM Repository is used by the all the different tools of the framework. For this purpose the tools assume a specific structure in the Repository that the developer should respect when uploading any source, input or description files.

As it is explained in the relevant documents (e.g. D1.4), the structure that the Repository follows contains a set of level-one folders which are considered the project directories. In the second level there is a set of directories defining the source (owner).



```
{
 "project" : "face-recognititon",
 "source"  : "development",
 "type":"Usercase",
 "content":"src_file",
 "any_user_label": "some value",
 "domain": "domain_hlrs"
}
```

```
{
 "project" : "face-recognititon",
 "source"  : "development",
 "type":"Usercase",
 "content":"src_file",
 "any_user_label": "some value",
 "domain": "domain_public"
}
```

Figure 14: Example of the Repository file – folder structure

The developer must upload all the necessary files on the Repository at *<ProjectName>/development* before the initiation of the tools. In specific, the following structure should be followed:

```
<ProjectName>/
    development/
        src/
            phantom_user_defined_structs.h
        description/
            Component-Network.xml
            Platform-Description.xml
        inputs/
        outputs/
```

4. Deployment Guides

In this section, guidelines about the individual deployment of each tool are provided with details that will ensure their installation and execution on any platform that satisfied the dependencies specified.

4.1 Repository

This section describes the deployment of the PHANTOM Repository and provides a usage guide. Please, keep in mind that these instructions may evolve fruit to extensions and improvements of the repository resulting from future work. Such updates will be available at [1]. Additionally, there are some video tutorials available at the PHANTOM YouTube channel [2].

The PHANTOM Repository is composed of two components: a web server and a data storage system. The web server provides various functionalities for data query and data analysis via RESTful APIs with documents in JSON format. The server's URL is "http://localhost:8000" by default. The default port for the https service is 8010.

4.1.1 Prerequisites

The server is implemented using Node.js, and connects to ElasticSearch to store and access Metadata. Before installing the required components, note that the installation and setup steps mentioned below assume a current Linux for an operating system. The installation was tested with Ubuntu 16.04 and 17.04 LTS. Before proceeding, cloning the repository is required:

```
git clone https://github.com/PHANTOM-Platform/Repository.git;
```

4.1.2 Dependencies

This project has the following dependencies:

Component	Homepage	Version
Elasticsearch	https://www.elastic.co/products/elasticsearch	= 2.4.6
Node.js	https://apr.apache.org/	>= 4.5
npm	https://www.npmjs.com/	>= 1.3.6

4.1.3 Installation

Before the installation of the Repository, it is required that ***npm*** be installed in advance. Once it is successfully installed, the Repository and the other required packages are automatically installed running one of the following scripts. The user can choose the appropriate shell scripts depending on their Operating System:

Shell script for Intel-x86 64bits (tested on Ubuntu):

```
bash setup-server-x86-64.sh;
```

Shell script for Intel-x86 32bits (tested on Ubuntu):

```
bash setup-server-x86-32.sh;
```

or the Shell script for Armv7l 64bits (tested on Raspbian):

```
bash setup-server-armv7-64.sh;
```

Notice that the script installs Nodejs 9.4.0. and Elastic-Search 2.4.6 on the folder {user_local_home_folder}/phantom_server. Details about the setup of ElasticSearch-database server can be found at [3][4][5][6].

The listening port is defined in the file repo_app.js, which default number is 8000, if the user wishes to use a different port, they can just modify it before starting or re-starting the Repository.

4.1.4 Start/Stop the server

Starting the PHANTOM Repository by ***executing*** the next script. For security reasons, the services may not start if they are requested from root.

```
bash start-repo.sh;
```

The server can be stop with the script:

```
bash stop-repo.sh;
```

In case of issues it can be verified that the Repository is running and has access to the database with the next scripts.

Test of the Nodejs Front-end running service:

```
curl http://localhost:8000;
```

Test if the Front-end has access to the ElasticSearch database Server.

```
curl -s http://localhost:8000/verify_es_connection;
```

To access to the Web-Interface, access with your internet browser to http://localhost:8000/repository.html

4.1.5 Configuration of USERS' accounts

After the installation, and before users can use the Repository, it is needed to register the users. This can be done using the script setup-new-server.sh provides an automatic method for register multiple users from the file list_of_users.ini.

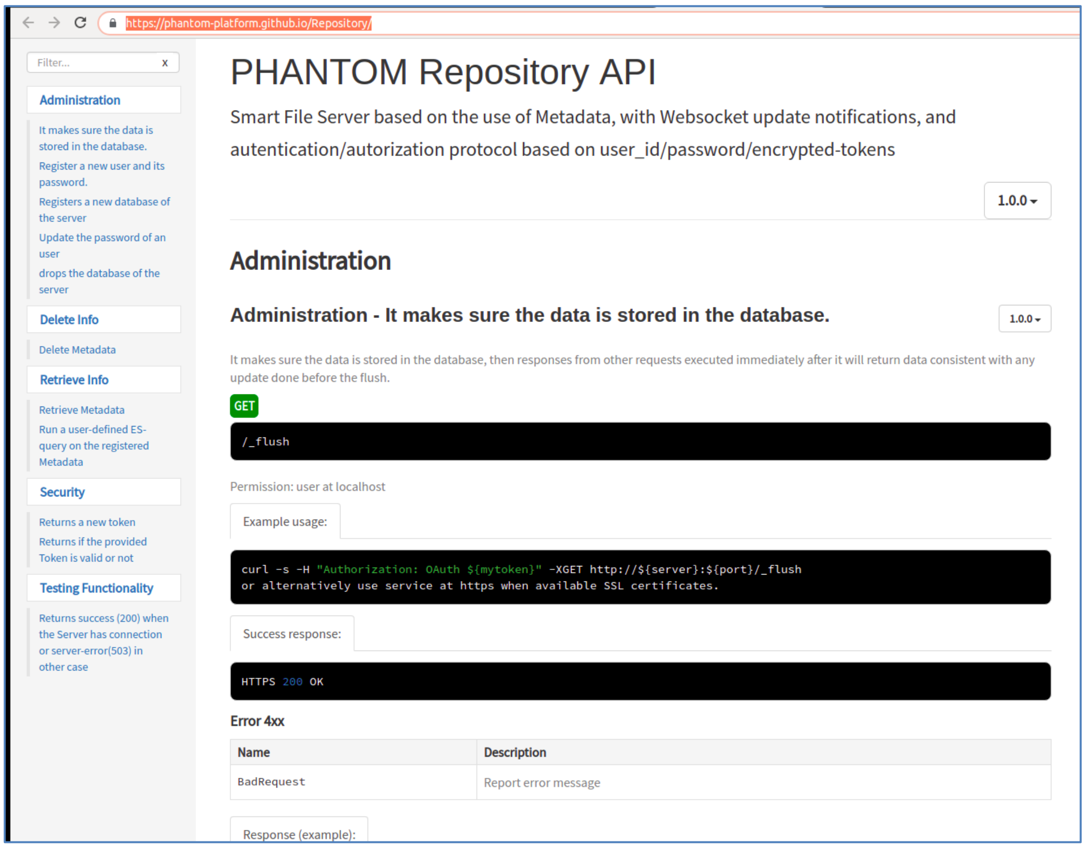
```
bash setup-new-server.sh;
```

NOTICE: For security reasons, users' accounts can be ONLY registered on the server. Requests from different IPs will be rejected.

For further details and examples are available in the folders api_command_line, api_bash_scripts, and api_java.

4.1.6 Reference manual of the RESTful API

The Reference manual of PHANTOM Repository RESTful API contains description of each of the implemented methods, with examples of use, and type of possible responses. The manual is built with the tool APIDOC, and formatted in a friendly HTML, where users can find a classification of the methods and a searching box. The manual is available online at https://phantom-platform.github.io/Repository/ (see next figure)


https://phantom-platform.github.io/Repository/
Filter...
x
Administration
It makes sure the data is stored in the database.
Register a new user and its password.
Registers a new database of the server
Update the password of an user
drops the database of the server
Delete Info
Delete Metadata
Retrieve Info
Retrieve Metadata
Run a user-defined ES-query on the registered Metadata
Security
Returns a new token
Returns if the provided Token is valid or not
Testing Functionality
Returns success (200) when the Server has connection or server-error(503) in other case
1.0.0
1.0.0
GET

PHANTOM Repository API

Smart File Server based on the use of Metadata, with Websocket update notifications, and autentication/autorization protocol based on user_id/password/encrypted-tokens

Administration

Administration - It makes sure the data is stored in the database.

It makes sure the data is stored in the database, then responses from other requests executed immediately after it will return data consistent with any update done before the flush.

```
/_flush
```

Permission: user at localhost

Example usage:

```
curl -s -H "Authorization: OAuth ${mytoken}" -XGET http://${server}:${port}/_flush
or alternatively use service at https when available SSL certificates.
```

Success response:

```
HTTPS 200 OK
```

Error 4xx

Name	Description
BadRequest	Report error message

Response (example):

Figure 15. Screenshot of the Online Reference Manual of the PHANTOM Repository API

4.2 Parallelisation Toolset (PT)

4.2.1 System requirements

A Linux distribution

A Linux distribution is required in order to support the installation of the ROSE compiler which is used by the tool. During development a Red Hat Enterprise Linux distribution has been used, as well as Ubuntu 16.04 LTS and Ubuntu 18.04 LTS, while other distributions can also be used as long as the ROSE Compiler can be successfully installed on the machine.

No more hardware requirements are specified.

4.2.2 Dependencies

Java 8

The jar executable file has been compiled using Java version 8, so a compatible version should be used for execution.

ROSE Compiler

A detailed guide for the installation of the ROSE Compiler and the *autoPar* tool that is used is provided in the following address:

http://rosecompiler.org/ROSE_HTML_Reference/installation.html

After the installation of ROSE, an executable of the autoPar tool will be available which should be able to be called by the PT internally. The PATH environmental variable should include the path to the autoPar executable, and LD_LIBRARY_PATH should include the path to the ROSE libraries.

4.2.3 Lite vs Full version

The simplified version of the Parallelization Toolset supports all the basic functionalities that are expected, namely:

- The support of the PHANTOM Programming Model.
- The interaction with all PHANTOM components such as the Repository, the Application and Execution Managers, the MOM, and the Deployment Manager.
- The support of library selection according to the selected Deployment Plan.

The features that are only supported by the full version are the following:

- Automatic generation of parallelized components

4.2.4 Deployment Procedure

An executable jar is needed for the execution of the component. The lite version can be directly downloaded from the following online repository:

https://github.com/PHANTOM-Platform/Parallelisation-Toolset

4.2.5 Configuration Guide

Along with the executable file, a configuration file (*"config.properties")* is available for the user to change the following fields:

- usermail: The username that is used to access the Repository.
- project: The name of the project.
- ipAddressRepo: Repository IP address

- portRepo: Repository port
- ipAddressApp: Application Manager IP address
- portApp: Application Manager port
- ipAddressExe: Execution Manager IP address
- portExe: Execution Manager port
- ipAddressMon: Monitoring Server IP address
- portMon: Monitoring Server port
- token: A valid token that can enable the use of the Repository for the requested project
- mode: < on | off > Enables/Disables parallelization. When off, the original versions of the components are adopted as the parallelized versions. An option to avoid the analysis time overhead that is caused.

4.2.6 Usage Guide

- **Execution steps**
  (run at terminal)

  a. *cd path/to/jarfile/*
  b. *java –jar ParallelizationToolset.jar*

- **Results**
  The results of the PT are updated versions of the source files and the component network. They are uploaded on the Repository.

4.2.7 Coordination with other components

The integration between the PHANTOM components is being coordinated by the Application Manager and the Repository servers. Thus, any integration aspects refer to the communication of the components via the use of the functionalities provided by these servers.

Code Analysis
In specific, the MOM expects the Code Analysis part of the PT to be completed before it continues with its execution. So, the PT is responsible to inform the Application Manager about the end of its execution so that the MOM can be initiated.

Technique Selection
Similar, to the MOM, Technique Selection listens to the Execution Manager for requested executions from the MOM (or the MBT or the user) in order to continue

with its functionality. When the module's execution is completed, a notification is sent to the Application Manager, so that the Deployment Manager can be initiated.

Repository
During execution, there are a lot of interactions with the Repository, for downloading and uploading files.

4.3 IP CORE GENERATOR

4.3.1 System requirements

The IP Core Generator requires a PC running Linux. It was tested on Ubuntu 16.04 LTS and Ubuntu 18.04 LTS, with AMD64 architecture, but should also run on any other Linux distribution as long as the dependencies are met.

4.3.2 Dependencies

- Xilinx Vivado tools with Zynq-7000 support (Tested with Vivado Design Suite 2016.4 and Vivado Design Suite 2017.4)
- Git
- Python3
- Python3 module: websocket-client

4.3.3 Deployment Procedure

The IP Core Generator needs Xilinx Vivado tools, with Zynq-7000 support, to be installed and properly configured. Vivado Design Suite comes with all the necessary tools and can be downloaded from:

https://www.xilinx.com/support/download.html

Installation and Licensing information can be found here:

https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0013-vivado-installation-and-licensing-hub.html

The Vivado Design Suite User Guide can be used as reference for detailed install instruction:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug973-vivado-release-notes-install-license.pdf

After installing Vivado Design Suite the tools must be imported into the current environment. It is necessary to source settings32.sh or settings64.sh (whichever is appropriate for the operating system) from the area in which the design tools are installed. This sets up the environment to point to this installed location. For example, for Vivado 2017.4, on the default install location:

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

Git and Python3 come pre-installed in most operating systems. If any of them is missing they can be installed with the following command:

```
sudo apt install git python3 python3-pip
```

Then the needed python modules can be installed using pip:

```
python3 -m pip install websocket-client
```

All the tool files can be downloaded to the current directory using git clone:

```
git clone https://github.com/PHANTOM-Platform/PHANTOM-IP-Core-Generator

git clone https://github.com/PHANTOM-Platform/IP-Core-Generator
```

Once everything has been installed, the tool can verify if the Xilinx tools are properly configured and accessible with:

```
./ipcore-generator.sh verify
```

4.3.4 Configuration Guide

The IP Core Generator will need to know how to connect to the Repository and App Manager. For this the corresponding IP addresses and ports should be set in the file *settings.py*. Also, the user must input their credentials to allow the IP Core Generator to connect to the other PHANTOM modules.

```
# Set the Repository IP address and port

repository_ip = "localhost"

repository_port = 8000


# Set the Application Manager websocket IP address and port

app_manager_ip = "localhost"

app_manager_port = 8500


# Set the credentials for the Repository and Application Manager

repository_user = "username"

repository_pass = "password"
```

Besides, IP addresses, ports and credentials, the only extra parameter that should be configured is the model of the target FPGA device, so the IP Core generator can target the correct hardware.

```
# Set the target FPGA device - ZC706 = xc7z045ffg900-2

target_fpga = "xc7z045ffg900-2"
```

4.3.5 Usage Guide

The IP Core Generator works autonomously, reading newly added files when it receives a notification from the Application Manager. To receive notifications from the Application Manager it is necessary to subscribe to a project.

```
./ipcore-generator.py  subscribe  project-name
```

The tool will then automatically run when new deployments are checked by the Offline MOM. If there is any component mapped to a FPGA the IP Core Generator will fetch the respective source files from the Repository to proceed with the analysis and source code transformation. It then calls Xilinx tools to generate an IP core, compresses it into a zip file and uploads it to the Repository.

The tool will also create a modified version of the software component that includes all the necessary code to interface between FPGA hardware and software automatically. The modified component will also be uploaded to the Repository.

Finally, the IP Core Generator will modify the component network to include the new FPGA implementation and update it in the Repository.

4.3.6 Manual usage

If you do not want to subscribe using the Application Manager, you can trigger the IP Core Generator to manually on a project using the remote command:

```
./ipcore-generator.py  remote  project-name
```

Finally, to avoid all dependencies on the Application Manager and Repository, you can simply run on a local PHANTOM project folder with the local command:

```
./ipcore-generator.py  local  /path/to/project/folder/
```

The outputs of the IP Core Generator (IP Core zip and modified component source code) will be stored in the ipcore-generator directory inside the project folder.

4.4 IP CORE MARKETPLACE

The IP Core Marketplace is an online repository where PHANTOM compliant IP Cores are stored. Users can check the IP Cores available and include or place existing functions by them.

4.4.1 Dependencies

For the IP Core Marketplace to be able to be used, it must be deployed first deployed on the Repository.

The recommended way to do it is by using the User-Scripts to download the IP Core Market and upload it to a predefined location on the repository. If the user wants to use this tool, then it have a dependency on User-Scripts.

Other way,the User only needs to use a way to send HTTP POST requests, like curl.

4.4.2 Installation Guide

The installation of the IP Core Marketplace with the User-Scripts can be done with a simple configuration on the 'settings.py', where the user needs to define the URL of were the git repository is hosted:

ipMarket_path = https://github.com/PHANTOM-Platform/PHANTOM-IP-Core-Marketplace.git

And the path where the IP Cores will be used store locally:

ip_folder = "IPCore-MarketPlace"

To perform deployment, the User needs to run:

```
demo@ubuntu:~/phantom-tools/User-tools$./start-PHANTOM.py -m
```

4.4.3 Usage Guide

To deploy an IP Core from the PHANTOM IP Core Marketplace, the developer must create a new implementation, in the Component Network, for the component which it is desired to incorporate the IP core. This component implementation will be specific to run on a Zynq, just like there are other component implementations specific for CPU or for GPU.

The component network implementation must be modified to include the path of IP core files and the FPGA software adapter source and header files, that are also provided by the Marketplace.

The developer should edit the component code and include the FPGA software adapter header file. Furthermore, the developer needs to replace the software function call with the function provided in the FPGA software adapter. The provided function will take care of interfacing with the FPGA hardware, copying the corresponding memory from the CPU memory region to the FPGA memory region, initializing the IP core, running it and copying back the results to the CPU memory.

This is all abstracted and the only effort needed from the developer, is adding a new implementation to the Component Network, including the provided header file and replacing the respective function calls on the component code.

4.5 MULTI OBJECTIVE MAPPER (MOM)

4.5.1 System requirements

No specific operating system is required and nor any hardware requirements are specified.

4.5.2 Dependencies

The jar executable file has been compiled using Java version 8, so a compatible version should be used for execution.

4.5.3 Lite vs Full version

The simplified version of the Multi-Objective Mapper supports all the basic functionalities that are expected, namely:

- The generation of deployment plans of parallel applications on the CPU-based hardware platforms.
- The support of the PHANTOM Programming Model.
- The interaction with all PHANTOM components such as the Repository, the PT, and the Monitoring framework.
- The support of requirements such as execution time, energy consumption, memory utilization and reliability.

The features that are only supported by the full version are the following:

- Mapping of components in GPUs and FPGAs.
- Multiple MOM iterations to provide improved performance deployment plans.
- Multiple MOM mutations to support extended design space exploration from the generated deployment plans.

4.5.4 Deployment Procedure

An executable jar is needed for the execution of the component. An executable jar is needed for the execution of the component. The lite version can be directly downloaded from the following online repository:

https://github.com/PHANTOM-Platform/GenericMOM

4.5.5 Configuration Guide

Along with the executable file, a configuration file (*"configuration.xml")* is available for the user to change the following fields:

- configuration name="GenericMOM"
- login value="LOGIN"
- password value="MYPASSWORD"
- ipAddress value="IPADDRESS"
- targetPort target="Repository" value="PORTNUMBER1"

- targetPort target="ApplicationManager" value=" PORTNUMBER2"
- targetPort target="ExecutionManager" value=" PORTNUMBER3"
- subscription type="project" value="MYPROJECT1"
- filepath value = "description"
- source value = "PT"
- target value = "MOM"
- componentNetworkFileName value = "cpn.xml"
- hardwarePlatformFileName value = "hw.xml"
- mfserveraddress value = " IPADDRESS "
- mfserverport value = " PORTNUMBER4"

4.5.6 Usage Guide

Execution steps

Run at terminal:

a. *cd path/to/jarfile/*
b. *java –jar MOM.jar -app "MOM_jar_file" -hw "component_network_XML file" -NMAP 10 -iMAP init_deployment.xml --online*

Arguments:
-jar "MOM_jar_file": indicating the name of MOM's executable jar file. Mandatory argument.
-app "component_network_XML file": indicating the name of the component network XML file. Mandatory argument when MOM does not access the remote repository.
-hw "hardware_platform_XML file": indicating the name of the hardware platform XML file. Mandatory argument when MOM does not access the remote repository.
-NMAP "number": indicating the number of the generated deployment plan population per MOM's genetic algorithm's iteration. Optional argument.
-iMAP "deployment_plan_XML file": indicating the name of the initial deployment plan XML file. Optional argument.
--manualData: flag which indicates MOM to use metrics provided by the user and not by the Monitoring Server or the MBT. Optional argument.
--online: flag which indicates MOM to access the remote repository as it is configured in the configuration file (see Prerequisite files below). Optional argument.
-project "project_name": indicating the specific target project name, overriding the projects subscribed in the configuration file. Optional argument.

Results

The generated mappings are uploaded to the Repository for the Deployment Manager to use them.

4.5.7 Coordination with other components

The integration between the PHANTOM components is being coordinated by the Application Manager, Execution Manager and the Repository servers. Thus, any integration aspects refer to the communication of the components via the use of the functionalities provided by these servers.

Parallelization Toolset
In specific, the MOM expects the Code Analysis part of the PT to be completed before it continues with its execution. So, the PT is responsible to inform the Application Manager about the end of its execution so that MOM can be initiated. Similarly, the Technique Selection expects a notification from the MOM (through the Application Manager) in order to continue with its execution.

Repository
During execution, there are a lot of interactions with the Repository, for downloading and uploading files.

4.6 OFFLINE MULTI OBJECTIVE MAPPER (OFFLINE MOM)

- **System requirements**

The OfflineMOM has no specific hardware requirements. It is tested to work on standard Linux distributions, but should also work on Windows and MacOS.

- **Dependencies**
  - Python 3
    - The OfflineMOM is written in Python 3.
  - Eclipse Epsilon
    - Translation, model checking, and model pattern matching, is implemented using the Epsilon framework, which is part of the Eclipse project.
  - MAST
    - The timing analysis is performed using the MAST tools from the University of Cantabria.

- **Deployment Procedure**

Python 3 probably comes with your operating system. If you are using MacOS and find you do not have Python 3, it is easy to install using Homebrew. Install Homebrew:

https://brew.sh/

and then issue the following install command:

```
brew install python3
```

It is already present in modern Linux distributions. For Windows, download and install from:

```
        https://www.python.org/
```

Epsilon should be downloaded from its website and unarchived to `/opt/eclipse`. If installed anywhere else, then set the environment variable `$ECLIPSE` to point to the installation directory.

```
        export ECLIPSE=/my/custom/install/location
```

MAST can be installed by downloading the latest binary release from:

```
        https://mast.unican.es/#downloading
```

This should then be unpacked and added to your system path. To check this is installed, the tool `mast_analysis` should be on your system path, or you can set the `$MASTEXE` variable.

```
        export MASTEXE=/mast/install/location/mast_analysis
```

You should also ensure that the PHANTOM Application Manager and Repository are correctly set up and running.

Once installed, the tool can be asked to check its dependencies:

```
        ./offlinemom.py verify
```

This will error if MAST or Epsilon cannot be found, and it will attempt to use credentials (see next section) to connect to the Repository.

- **Configuration Guide**

  The only configuration required for the OfflineMOM is that it needs credentials to connect to the Repository and Application Manager.

  When you first run the application, it will try to read credentials.txt. Unless you have already created it specifically, this will fail, and it will be autogenerated with default values. You should edit this file with the username, password, and TCP port of the PHANTOM Repository.

- **Usage Guide**

  To start the Offline MOM, it should be told to subscribe to an Application Manager project. To subscribe to a project called projectname use the following:

```
        ./offlinemom.py subscribe projectname
```

The tool will connect to the Application Manager and subscribe. It will then automatically run when new deployments are added.

It is intended that the developer does not manually interact with the OfflineMOM. It communicates through the metadata of deployments in the PHANTOM Repository. Files of type "deployment" will have a metadata variable "checked". If this is not set to "yes" then the OfflineMOM will download it, check it, and set "checked" to "yes" if it passed all checks.

If a deployment failed at least one check, then "checked" will be set to the name of the failing check, and the schedulability report will be uploaded to the repository in the same location as the deployment. For example: if the deployment failed on the fixedpriority1 check, then fixedpriority1.txt will be uploaded.

- **Manual usage**

If you do not want to subscribe using the Application Manager, you can trigger an analysis manually on a project using the remote command:

```
./offlinemom.py remote projectname
```

The results will be output to the console.

Finally, to avoid all dependencies on the Application Manager and Repository, you can simply run on a folder of PHANTOM XML files with local:

```
./offlinemom.py local /path/to/folder/
```

4.7 DEPLOYMENT MANAGER (DM)

- **System requirements**

The tools can be evenly executed on both Linux and Windows platforms. However, the produced scripts that enable the compilation and deployment of the application can only be executed on a Linux environment (a Red Hat Enterprise Linux distribution has been tested, as well as Ubuntu 16.04 LTS and Ubuntu 18.04 LTS).

- **Dependencies**

During deployment the tool interacts with the PHANTOM Repository, so it is assumed that the application is uploaded in the corresponding location, as well as the files from preceding tools like MOM and the PT.

- **Deployment Procedure**

An executable jar is needed for the execution of the component. The lite version can be directly downloaded from the following online repository:

https://github.com/PHANTOM-Platform/Deployment-Manager

- **Configuration Guide**

Along with the executable file, a configuration file (*"config.properties")* is available for the user to change the following fields:

- usermail: The username that is used to access the Repository.
- project: The name of the project.
- ipAddressRepo: Repository IP address
- portRepo: Repository port
- ipAddressApp: Application Manager IP address
- portApp: Application Manager port
- ipAddressMon: Monitoring Server IP address
- portMon: Monitoring Server port
- ipAddressExe: Execution Manager IP address
- portExe: Execution Manager port
- token: A valid token that can enable the use of the Repository for the requested project

- **Usage Guide**

Execution steps

1. Run at terminal:
   a. *cd path/to/jarfile/*
   b. *java –jar DeploymentManager.jar*
2. The process from here on is automated.
   a. The components are refined and the refined versions are uploaded on the Repository
   b. The Makefile is refined and uploaded on the Repository
   c. The necessary scripts for the compilation and deployment of the application are produced.
3. Application execution:
   a. Automatic execution
   b. Manual execution: *bash deploy<execution_id>*.sh

The execution of the application starts.

- **Results**

Compilation and execution scripts generation
The scripts necessary for the compilation and the execution of the application are generated. Those are found in /path/to/jarfile.

- **Coordination with other components**

Parallelization Toolset
The Deployment Manager expects a notification from the PT (through the Application Manager) in order to continue with its execution. The notification informs the DM that the necessary modifications in the source files have been made and uploaded on the Repository, so the deployment procedure can commence.

Repository
During execution, there are a lot of interactions with the Repository, for downloading and uploading files.

4.8 PHANTOM FPGA Linux Software Distribution

The PHANTOM Linux software distribution contains scripts to build a full Linux environment for Zynq-7000 SoCs based on a platform definition.

- The built platform includes:
- Linux kernel
- Linux root file system (either BusyBox or Debian/Ubuntu)
- Bootloaders (Zynq FSBL and U-Boot)
- Zynq boot image
- FPGA bitstream
- Customised Linux device tree
- PHANTOM communications API (including Open MPI)
- PHANTOM component definition XML file

Prebuilt images are provided for the Xilinx ZC706 board, which just require copying to an SD card in order to boot the system.

4.8.1 Installation

To begin, clone the repository:

```
git clone https://github.com/PHANTOM-Platform/ ¥
PHANTOM-FPGA-Linux.git
```

4.8.1.1 Prerequisites

Before running the build script you will need:

- Multistrap (if using the Debian-based file system)
- The Device Tree Compiler (dtc)
- mkimage
- libssl
- QEMU
- Python 3
- Xilinx Vivado tools with Zynq-7000 support (tested with version 2018.2)

On Debian or Ubuntu-based distributions you can install these with the following command:

```
sudo apt-get install multistrap device-tree-compiler u-boot-tools libssl-dev dpkg-dev qemu-user-static python3
```

To install the Xilinx tools, consult the documentation that comes with Vivado. The tools must be imported into the current environment, so that the vivado, hsi and bootgen commands are runnable from the command line.

4.8.1.2 Quick start with prebuilt images

The `prebuilt` folder contains a ready-built set of images that can be used to boot a Xilinx ZC706 board, including a default set of dummy components on the FPGA logic and in the Linux device tree, and a BusyBox-based root file system.

To create a system with these images, first, copy the included prebuilt kernel, file system, bitstream and boot images to be used on the board:

```
./make.sh prebuilt
```

Next, format an SD card with a FAT32 file system of at least 30MB, and ensure it is mounted at `/media/$USER/BOOT`.

To copy the images to the SD card, run:

```
./make.sh sdcard
```

Insert this SD card into the ZC706 board, set the boot select switches (SW11) to 0-0-1-1-0 for SD boot, and turn on the board with a console connected to the USB UART at 115200 bps. Once booted, login with user "root" and password "phantom". The FPGA components should be accessible at `/dev/phantom/`.

4.8.1.3 Quick start with custom configuration

Before using the build scripts, you must create a configuration describing which FPGA board to target, which root file system type to use, and which FPGA components to include in the design.

These options are set in the `phantom_fpga_config.json` file, with the following format:

```
{
 "target": {
       "board": "board_name",
       "rootfs": "rootfstype"
 },
 "ipcores": [
       {
             "ipname": "vendorname:libraryname:ipcore:1.0",
             "memory": 4096
       }
 ]
}
```

- `target` describes the deployment target of the design being generated, as follows:
  - `board` should be set to the target board type, as defined in `boardsupport.sh` (e.g. `zc706`, `zybo`, `zedboard`)
  - `rootfs` should be set to the desired root file system type, either `buildroot` or `multistrap`
- `ipcores` should contain a list of the IP cores to include in the design, along with their shared memory requirements, as follows:
  - `ipname` is the name of a PHANTOM IP core available in `arch/phantom_ip/`, as recognised by Vivado (the standard format of this field in Vivado is `vendor:library:name:version`)
  - `memory` is the amount of shared memory (in bytes) to reserve for access by the IP core's master interface and associated Linux driver. The build

scripts will round this number to the next power of two, and at least 4KiB. A value of `0` means no shared memory will be available.

4.8.1.4 Building the hardware project

Ensure your PHANTOM-compatible IP cores (see later) are in `arch/phantom_ip/` and run the following:

```
./make.sh hwproject

./make.sh implement
```

4.8.1.5 Multistrap (Debian-based) root file system

If using the Debian-based root file system, set `rootfs` to `multistrap` in `phantom_fpga_config.json` and generate with:

```
./make.sh rootfs
```

The script will ask for root permissions after downloading packages, to allow it to chroot into the new file system to complete package set-up and set the root password (the user will be prompted for this).

If Linux kernel modules or Open MPI libraries are required in the file system, these must be built beforehand so they can be copied in. Therefore, for a complete file system, run the following:

```
./make.sh sources

./make.sh kernel

./make.sh ompi

./make.sh rootfs
```

Note: if you get unusual errors whilst compiling, (such as that the compiler is not C and C++ link compatible) ensure that you have sourced Xilinx's setup scripts and that you are therefore compiling using their toolchain.

4.8.1.6 Set up an SD card (or alternative storage device)

If using the BusyBox-based root file system, the images can be copied directly to flash memory (using a third-party tool), or to a single FAT partition on an SD card, using the instructions below.

If using the Debian-based file system, an SD card (or similar storage) is required to hold the boot images and root file system on separate partitions.

Format an SD card with two partitions:

1. The first, a small FAT32 partition called BOOT. This is just to hold the bootloaders, kernel, and FPGA bitstream, so 30MB is typically plenty of space.

2. The rest of the card as an ext4 partition called Linux.

Ensure that the SD card partitions are mounted and that the SDCARD_BOOT and SDCARD_ROOTFS variables at the top of make.sh are correctly set.

Finally copy all boot files and the root file system to the SD card, using:

./make.sh sdcard

The FPGA board can now be programmed and booted to Linux using this SD card.

4.8.2 PHANTOM-compatible IP Cores

The architecture scripts build an FPGA design from a set of PHANTOM-compatible IP cores in `arch/phantom_ip/`. A PHANTOM-compatible IP core has the following characteristics:

- Exactly one AXI Slave interface, which is used to control the core via UIO-mapped registers.
- Zero or more AXI Master interfaces which are used for high-speed access to main memory.
- An optional interrupt line for triggering interrupt handlers in Linux userland (not currently implemented).

The IP core should also be an IP core as generated by the Xilinx tools (such as from Vivado HLS or packaged by Vivado).

4.8.3 Building images from sources

4.8.3.1 Setting-up board support and build variables

To build the images, the first task is to ensure the target board is defined in `boardsupport.sh`, along with appropriate build variables.

To support a non-default board, the following variables should be used in `boardsupport.sh`, copying the format of existing entries:

- `DEVICETREE` should be the name of the device tree in the Linux kernel tree to use. Xilinx provides these for all of its boards in the `arch/arm/boot/dts/` folder of the kernel source.
- `UBOOT_TARGET` should be the target board to build U-Boot for. The available configurations are in the `configs`directory of the U-Boot source.
- `BOARD_PART` should be the Xilinx name for the target board. You can list all of the board parts that your Xilinx installation supports by entering the command `get_board_parts` into the TCL console of Vivado.

The `VIVADO_VERSION`, `OMPI_VERSION` and `BUILDROOT_VERSION` variables can be customised to match the desired source versions to download and build. In particular, `VIVADO_VERSION` should be set to match the version of Vivado used to build the hardware. The default Vivado version is `2018.2`.

If any extra customisation is needed to the Linux kernel build, configuration parameters can be added to the `custom/kernel_config` file, whose contents will be appended to the default config (`xilinx_zynq_defconfig`) before the kernel is built.

The board type to use when building the system should then be set in `phantom_fpga_config.json` (see above).

4.8.3.2 Building U-Boot and the Linux kernel

Once the board is defined, the Linux kernel and U-Boot sources can be downloaded and built with the following:

```
./make.sh sources

./make.sh uboot

./make.sh kernel
```

These commands also copy the built products to the `images/` folder. The U-Boot runtime environment is generated separately based on the specific FPGA hardware design, and can be found in `images/uEnv.txt` after the hardware project is created.

4.8.3.3 Creating an FPGA hardware design

The PHANTOM distribution also contains the scripts that create PHANTOM-compatible FPGA designs. A PHANTOM hardware design encapsulates a set of IP cores, makes them available to the software running in the Linux distribution, and includes the various security and monitoring requirements of the PHANTOM platform.

To build a hardware project, first check ensure that the IP cores you are using are in the `arch/phantom_ip/` directory. This directory already contains two dummy IP cores, which can be used for testing.

Next, edit `phantom_fpga_config.json` to describe the specific hardware design requirements, including FPGA board type, the IP cores to include, and the shared memory requirements of those IP cores (see above for a description of the file structure).

Once set, execute the following to create the hardware project and then perform Vivado implementation on the design to produce a bitstream:

```
./make.sh hwproject

./make.sh implement
```

The resulting hardware project will be created in the `hwproj/` directory. Alongside the hardware project itself, the scripts will generate a matching PHANTOM component definition XML file, Linux device tree overlay describing the hardware, and a compatible U-Boot environment definition, all output to `images/`.

4.8.3.4 Device tree generation

You must have a suitable device tree for U-Boot and the Linux kernel to work on your target board. Xilinx's Linux kernel repository contains device trees for many boards in the `arch/arm/boot/dts/` folder. These all reference a base tree called `zynq-7000.dtsi` which describes the generic Zynq SoC architecture. If your target board requires a custom device tree, ensure it is copied into the kernel and U-Boot source tree and matches the associated definitions in `boardsupport.sh`.

In order to leave the board's Linux kernel device tree untouched, PHANTOM components are described in a device tree *overlay*, which is dynamically applied to the base device tree on each boot by U-Boot. The build scripts create this device tree overlay based on the PHANTOM component definition XML output from the hardware project creation. See `arch/generate_environment.py` for how this overlay is generated.

The base device tree and device tree overlay are generated when building the kernel and hardware project respectively, but if required they can be built separately using:

```
./make.sh devicetree
```

4.8.3.5 Generating an FSBL

An FSBL (first stage bootloader) is required to start the boot process, and sets up various components of the Zynq-7000 device.

You can generate an FSBL based on the current hardware design and board type, using:

```
./make.sh fsbl
```

This will create `images/fsbl.elf`. Alternatively, an FSBL can be created using Xilinx SDK.

4.8.3.6 Creating a boot image

The FSBL and U-Boot must be combined into a single boot image in order to boot a board from an SD card.

After the FSBL and U-Boot executables are generated, of if they change, the boot image can be created using:

```
./make.sh bootimage
```

This will create `images/BOOT.bin`. Alternatively, a boot image can be created using Xilinx SDK.

4.8.3.7 Building Open MPI

Open MPI can be downloaded and built for Linux on the Zynq using the build scripts, and will be installed to `/opt` on the created root file system by default.

Set the `OMPI_VERSION` variable in `boardsupport.sh` as required (the default is to use v3.0.0). If needed, the download URL can also be customised by changing `OMPI_URL`.

Open MPI can then be built and installed with the following:

```
./make.sh sources

./make.sh ompi
```

Open MPI must be built *before* creating the root filesystem, if it is to be included.

4.8.3.8 Creating a root file system

The make script can create either a Debian-based root file system using Multistrap, or a BusyBox-based root file system using Buildroot. The Debian file system is designed to be mounted as the system's main persistent storage (e.g. from an SD card), whereas the BusyBox system is better suited to running as an ephemeral RAM disk.

The file system can be generated by setting the `rootfs` type in `phantom_fpga_config.json` to either `multistrap` or `buildroot`, then running:

```
./make.sh sources # (if using Buildroot)

./make.sh rootfs
```

If the appropriate sources have been downloaded and built beforehand, this will also copy Open MPI, Linux kernel modules and the PHANTOM API libraries into the file system.

Alternative Linux file systems can be used, but are not supported by these scripts.

Buildroot file system customisation

The Buildroot-generated file system can be modified using the configuration file, post-build script and file system overlay in the `buildroot-phantom` folder.

More information is available in the Buildroot manual.

Multistrap file system customisation

The basic contents of the file system can be customised by editing `multistrap/multistrap.conf` before building. This file defines the packages included, as well as the Debian version to use (both Debian 8 (Jessie) and 9 (Stretch) should work). The default configuration uses Debian 9 (Stretch), and includes a selection of useful packages for a fairly full-featured system.

As an alternative to Debian, an optional Ubuntu 18.04 LTS (Bionic Beaver) configuration is also included, in `multistrap/multistrap-ubuntu.conf`. To use this, replace `multistrap/multistrap.conf` with this file.

The `multistrap/rootfs_setup.sh` script is run to set-up the Multistrap system after packages have been downloaded. This file can be modified to customise this process.

Additional files can be added to the root file system automatically by the make script by placing them in the `multistrap/overlay/` folder.

4.8.3.9 File system size

The Debian root file system created by the scripts is designed to be copied to an SD card and mounted as the system's main storage, so can be quite large.

The following are estimated sizes for the built file system, where 'complete' is the full default included `multistrap.conf`and 'minimal' is only the base Debian packages required for booting:

- Jessie (complete) - 388MB
- Jessie (minimal) - 186MB
- Stretch (complete) - 395MB
- Stretch (minimal) - 169MB

This includes around 13MB for Open MPI, kernel modules and the PHANTOM API on top of the Debian system.

The compressed image of the BusyBox file system is around 10MB by default (with Open MPI, kernel modules and PHANTOM API included).

4.8.4 Support files for additional boards, etc.

The `support/` folder contains a range of additional files that can be installed for working with non-standard boards, as well as potentially useful kernel patches and optional configuration options.

See `support/README.md` for more information.

4.8.5 Running a full build from sources

The following series of make script commands will run a typical full build and install of all components from sources. This is equivalent to `./make.sh all`.

The contents of `phantom_fpga_config.json` should be set before starting the build process, and the SD card partitioned and mounted ready for use.

If any Linux kernel, U-Boot or Buildroot customisations are required (patches, overlays, etc.), these should be applied after fetching sources but before building these components.

```
./make.sh sources
./make.sh hwproject
./make.sh implement
```

```
./make.sh fsbl
./make.sh uboot
./make.sh bootimage
./make.sh kernel
./make.sh ompi
./make.sh rootfs
./make.sh sdcard
```

4.9 MONITORING FRAMEWORK – SERVER

The PHANTOM Monitoring Server is one part of the PHANTOM Monitoring Framework, which supports monitoring of performance and power metrics for heterogeneous platforms and hardware.

4.9.1 Introduction

The PHANTOM Monitoring Server is composed of two components: a web server and a data storage system. The web server provides various functionalities for data query and data analysis via RESTful APIs with documents in JSON format. The server's URL are "http://localhost:3033" by default. The default port for the https service is 3043.

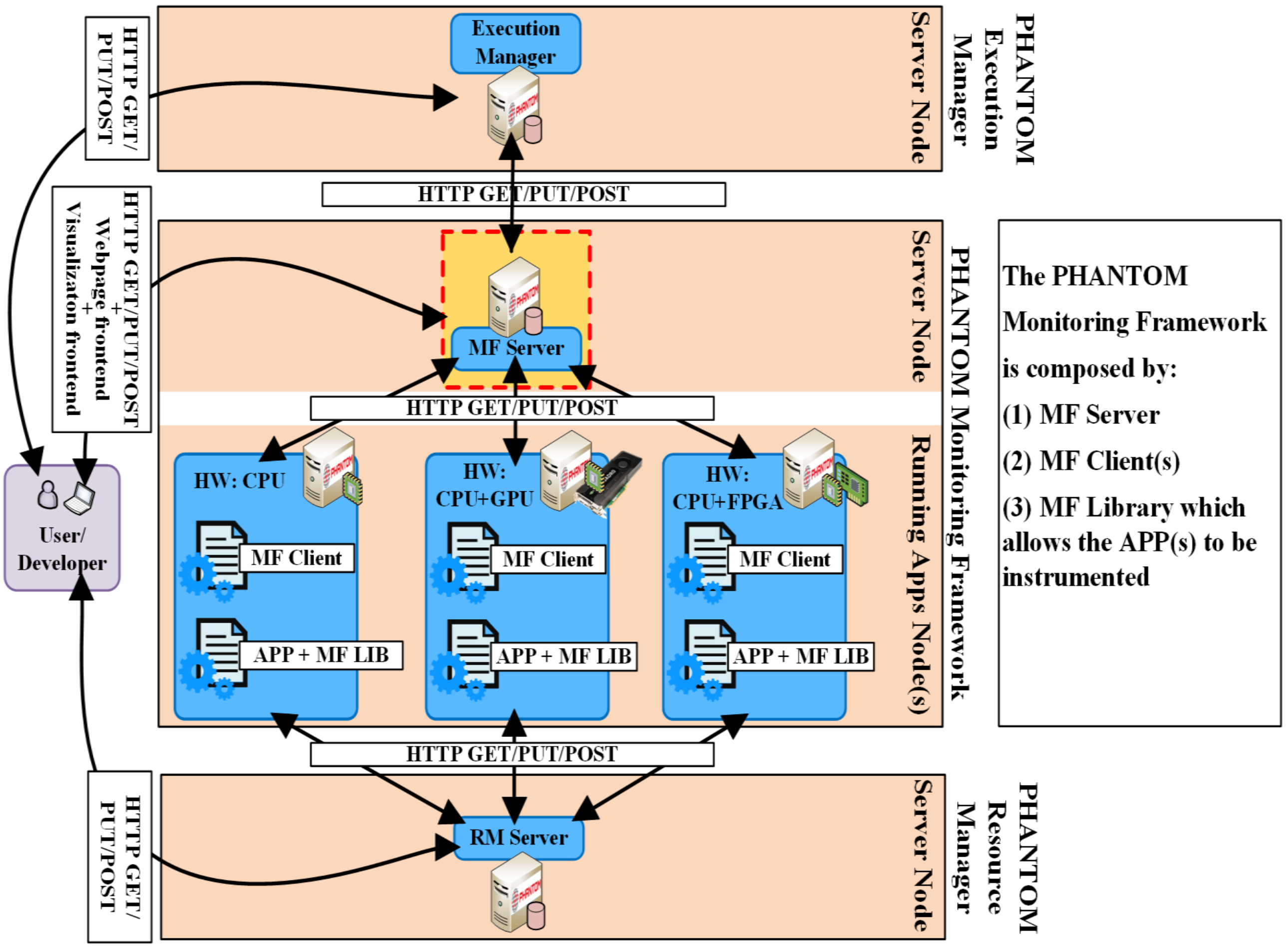


Figure 16: Monitoring Server

4.9.2 Prerequisites

The Monitoring Server receives data from the PHANTOM monitoring client via RESTful interfaces. The server is implemented using Node.js, and connects to Elasticsearch to store and access metric data. Before you start installing the required components, please note that the installation and setup steps mentioned below assume that you are running a current Linux as operating system. The installation was tested with Ubuntu 16.04 LTS as well as with Scientific Linux 6 (Carbon). Before proceeding, cloning the repository is required:

```
svn export https://github.com/PHANTOM-
Platform/Monitoring.git/trunk/Monitoring_server Monitoring_server
```

4.9.3 Dependencies

This project requires the following dependencies to be installed:

Component	Homepage	Version
Elasticsearch	https://www.elastic.co/products/elasticsearch	= 2.4.6
Node.js	https://apr.apache.org/	>= 4.5
npm	https://www.npmjs.com/	>= 1.3.6

4.9.4 Installation

This section assumes that all required dependencies have been successfully installed as described in the previous paragraphs.

Shell script for Intel-x86 32bits (tested on Ubuntu):

```
bash setup-server-x86-32.sh
```

or the Shell script for Intel-x86 64bits (tested on Ubuntu):

```
bash setup-server-x86-64.sh
```

or the Shell script for Armv7l 64bits (tested on Raspbian):

```
bash setup-server-armv7-64.sh
```

This setup provides the configuration to the elastic-search by placing in the appropriate folder the file elasticsearch.yml (from this repository). That configuration file makes that ElasticSearch uses the port 9400 instead of the default port 9200.

4.9.5 Start/Stop the server

Starting the PHANTOM Monitoring Server by ***executing*** the next script. For security reasons, the services may not start if they are requested from root.

```
bash start-monitoring-server.sh
```

The server can be stop with the script:

```
bash stop-monitoring-server.sh
```

In case of issues it can be verified that the Monitoring Server is running and has access to the database with the next scripts.

Test of the Nodejs Front-end running service:

```
curl http://localhost:3033;
```

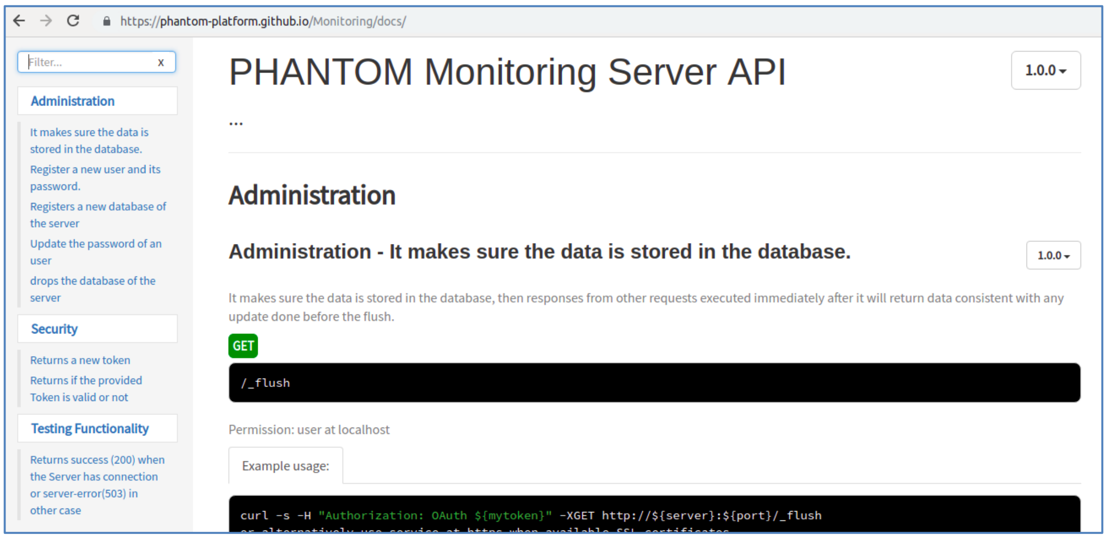
Test if the Front-end has access to the ElasticSearch database Server.

```
curl -s http://localhost:3033/verify_es_connection;
```

To access to the Web-Interface, access with your internet browser to http://localhost:3033/monitoringserver.html

4.9.6 Reference manual of the RESTful API

The Reference manual of PHANTOM Monitoring Server RESTful API contains description of each of the implemented methods, with examples of use, and type of possible responses. The manual is built with the tool APIDOC, and formatted in a friendly HTML, where users can find a classification of the methods and a searching box. The manual is available online at https://phantom-platform.github.io/Monitoring/docs/ (see next figure).



Administration

It makes sure the data is stored in the database.

Register a new user and its password.

Registers a new database of the server

Update the password of an user

drops the database of the server

Security

Returns a new token

Returns if the provided Token is valid or not

Testing Functionality

Returns success (200) when the Server has connection or server-error(503) in other case

PHANTOM Monitoring Server API

...

Administration

Administration - It makes sure the data is stored in the database.

It makes sure the data is stored in the database, then responses from other requests executed immediately after it will return data consistent with any update done before the flush.

GET

```
/_flush
```

Permission: user at localhost

Example usage:

```
curl -s -H "Authorization: OAuth ${mytoken}" -XGET http://${server}:${port}/_flush
```

Figure 17. Screenshot of the Online Reference Manual of the PHANTOM Monitoring Server API

4.10 MONITORING FRAMEWORK – CLIENT

The PHANTOM Monitoring Client is one part of the PHANTOM Monitoring Framework, which supports metrics collection based on hardware availabilities and platform configuration. The target platform of the monitoring client includes CPUs, GPUs, Myriad2, ACME power measurement kit and FPGA-based platform.

4.10.1 Introduction

The essential functionality of PHANTOM monitoring client is metrics collection, which is implemented by various plug-ins according to the hardware accessibility.

Currently 7 plug-ins are supported, whose implementation and design details are collected in the directory `src/plugins`. The monitoring client is designed to be pluggable. Loading a plug-in means starting a thread for the specific plug-in based on the users' configuration at run-time. The folder `src/agent` plays a role as the main control unit, as managing various plug-ins with the help of pthreads. Folders like `src/core`, `src/parser`, and `src/publisher` are used by the main controller for accessorial support, including parsing input configuration file (`src/mf_config.ini`), publishing metrics via HTTP, and so on.

For code instrumentation and user-defined metrics collection, we provide a monitoring library and several APIs, which are kept in the directory `src/api`. Descriptions in detail about how to use the monitoring APIs are given also in this directory.

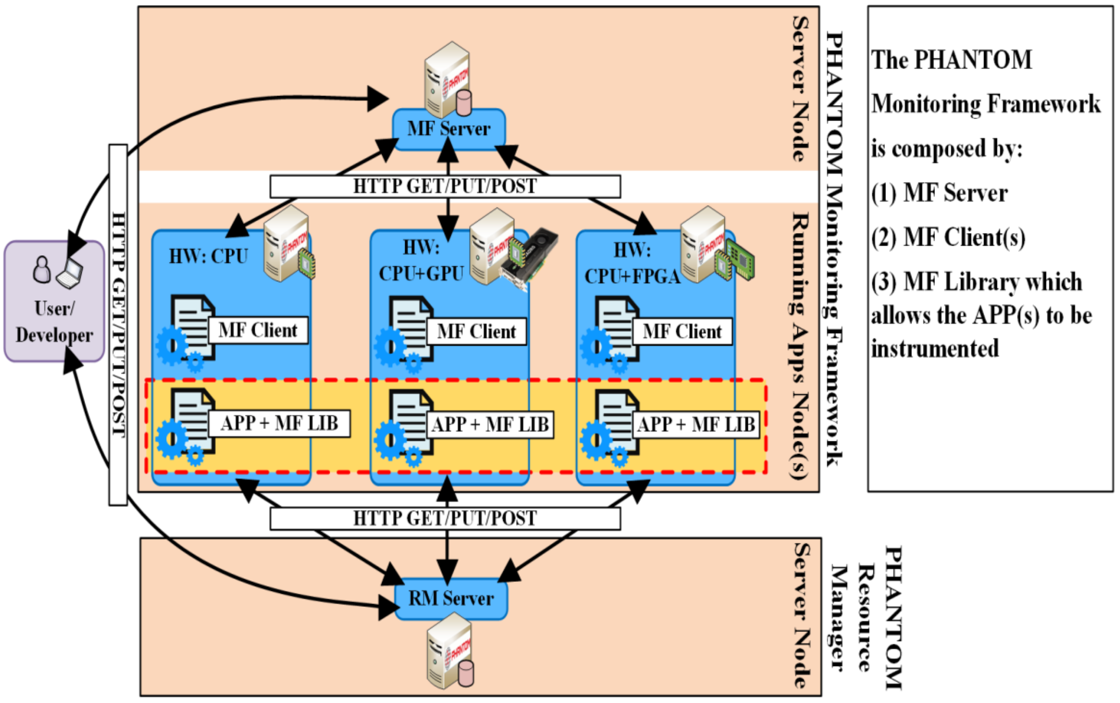


Figure 18: Monitoring Client

4.10.2 Prerequisites

The Monitoring Client requires at first a running server and database. In order to install these requirements, a checkout of the associated PHANTOM monitoring server and follow the setup instructions given in the repository `README.md` file. A successful setup process can be checked by the following command as testing whether the server is running in the specific url:

```
curl http://localhost:3033;
```

Note that the installation and setup steps mentioned below assume that you are running a current Linux as operating system. We have tested the monitoring agent with Ubuntu 14.04 LTS as well as with Scientific Linux 6 (Carbon).

Before proceeding, cloning the repository is required:

```
svn export https://github.com/PHANTOM-
Platform/Monitoring.git/trunk/Monitoring_client Monitoring_client
```

4.10.3 Dependencies

This project has the following dependencies:

Component	Homepage	Version
PAPI-C	http://icl.cs.utk.edu/papi/	5.4.0
CURL	http://curl.haxx.se/download/	7.37.0
Apache APR	https://apr.apache.org/	1.6.3
Apache APR Utils	https://apr.apache.org/	1.6.1
Nvidia GDK	https://developer.nvidia.com/gpu-deployment-kit/	
for 64 bits system		352.55
for 32 bits system		340.29
bison	http://ftp.gnu.org/gnu/bison/	3.0.2
flex	http://prdownloads.sourceforge.net/flex/	2.6.0
sensors	https://fossies.org/linux/misc/	3.4.0
m4	https://ftp.gnu.org/gnu/m4	1.4.17
libiio	https://github.com/analogdevicesinc/libiio.git	1.0
hwloc	https://www.open-mpi.org/software/hwloc/v1.11/downloads/	1.11.2
EXCESS queue	https://github.com/excess-project/data-structures-library.git	release/0.1.0

Notice: The version of the packages are the same for 32 and 64 bits systems, except for the version of Nvidia GDK.

To ease the process of setting up a development environment, we provide a basic script that downloads all dependencies, installs them locally in the project directory, and then performs some clean-up operations. Thus, compiling the monitoring client can be performed in a sandbox without affecting your current operating system.

Executing the next script if your device has a 32 bits Operating System:

```
bash setup-client-32.sh;
```

OR Executing the next script if your device has a 64 bits Operating System:

```
bash setup-client-64.sh;
```

results (in any of these cases) in a new directory named `bin`, which holds the required dependencies for compiling the project.

4.10.4 Installation

This section assumes that you've successfully installed all required dependencies as described in the previous paragraphs.

```
make clean-all
make all
make install
```

The above commands clean, compile and install the monitoring agent into the directory `dist` within the project's repository. The `dist` folder includes all required binaries, shared libraries, scripts, and configuration files to get you started. The Makefile has been tested with GNU compiler version 4.9.2.

4.10.5 Start monitoring

If you haven't yet followed our guide to set up the associated monitoring server and database, please do so now before continuing. Next, start the monitoring client with a default set of plugins enabled to monitor as follows:

```
cd scripts
./start.sh
```

You can learn more about various options passed to the monitoring client by calling

```
./start.sh -h
```

While the monitoring client is started and is collecting metric data, you can use the RESTful APIs provided by the monitoring server to retrieve run-time metrics and corresponding statistics.

4.10.6 Configuring plug-ins and update intervals

The Monitoring Client as well as plug-ins are configurable at run-time by a global configuration file named `mf_config.ini`. The configuration is implemented by using an INI file; each section name such as `timings` or `plugins` is enclosed by square brackets. For each section, various parameters can be set. These parameters are custom-defined for each plug-in.

```
;PHANTOM Monitoring Client Configuration

[generic]
server = http://localhost:3033/v1
...
```

```
[plugins]
mf_plugin_Board_power = on
mf_plugin_CPU_perf = off
...

[timings]
default               = 1000000000ns
update_configuration  = 360s
mf_plugin_Board_power = 1000000000ns
...

[mf_plugin_Board_power]
ACME_BOARD_NAME = baylibre-acme.local
device0:current = on
device0:vshunt = off
device0:vbus = off
device0:power = on
...
```

Several parameters such as the `timing` of the plug-ins or the `server` where the server is running can be configured through this configuration file. The file is called `mf_config.ini` and is located at `dist/mf_config.ini`.

4.11 SECURITY MANAGER

The scope of the PHANTOM security effort was delineated in Section 9 of the deliverable D1.2 – First design for Cross-layer Programming, Security and Runtime monitoring. This section describes the deployment of the PHANTOM Security Manager. Discussed here is primarily the access control system based on the Next Generation Access Control standard.[11] This aspect of PHANTOM security is implemented by distinct security components. The execution integrity aspect of PHANTOM security is achieved through the architectural combination and features of the components of the PHANTOM framework that enable a component network to be transparently and effectively realised.

- **Introduction**

The PHANTOM Security Manager access control system is composed of two components, the 'ngac' Policy Tool and the 'ngac-server' Policy Server. The Policy Tool is used to test NGAC policies during their development. The Policy Server uses those policies to provide a runtime policy decision making service. These components share a common set of Prolog source modules and their separate executables are constructed with a simple shell script, ***mkngac***, which accompanies the sources. A primary objective of the 'ngac'/'ngac-server' development is to create a lightweight and highly portable access control framework that can be easily adapted to different situations and applications, that has a minimum of external dependencies, and that requires minimal resources to run. This objective has been achieved to a high degree in the current implementation. The design, features, and API and user interfaces of the NGAC implementation, and of the design of execution integrity through isolation and communication control, are described in Section 2.7 of the deliverable D1.4 – Final design for Cross-layer Programming, Security and Runtime monitoring.

The 'ngac' Policy Tool is used to do standalone development and testing of policy files. The 'ngac-server' Policy Server is integrated with the PHANTOM Repository, which has modules that act as the NGAC Policy Enforcement Point (PEP) and Resource Access Point (RAP) of the NGAC Functional Architecture, as described in Section 2.3 of the deliverable D5.2 – Integrated reference system.

- **Prerequisites**

The 'ngac' Policy Tool and the 'ngac-server' Policy Server are implemented in the Prolog language and require the SWI-Prolog environment to run. The software is provided as a set of Prolog source files and/or as "executable" files that have the Prolog runtime environment already linked in.

SWI-Prolog is available for several operating environments, including Microsoft Windows (64 bit) and (32 bit), MacOS X 10.6 and later on Intel, and several Linux versions including Ubuntu. It is also available in Docker containers and as a source distribution that one can build locally.

Our NGAC implementation uses *only* the libraries that come with the Prolog distribution. Furthermore we've organised the functional architecture so as to place adaptation into the hands of application developers without requiring modification to the core implementation. This is in contrast to the reference implementations that have so many external dependencies (some obsolete) so as to be very cumbersome to work with and not very portable or adaptable.

To begin, clone the GitHub repository for the Policy Tool and Server located at:

https://github.com/PHANTOM-Platform/Security-Server

- **Dependencies**

This project has the following sole dependency:

Component	Homepage	Version
SWI Prolog	www.swi-prolog.org	= 7.6.4

- **Installation**

Prior to installation or building of the Security Manager it is required that ***swipl*** be installed. The exception to this requirement is if there is already a compiled version of 'ngac' and 'ngac-server' for the target platform, in which case all the dependencies are already linked into the executable.

The software consists of a directory that includes source files and the executable files, ***ngac*** and ***ngac-server***. It is advisable to remake the executables since if your system differs from the distribution system where the executables were built, the executables are not likely to run. The executables for the 'ngac' Policy Tool and the 'ngac-server' Policy Server are made by the shell script ***mkngac***, located with the source files, that must be run in an environment that has SWI-Prolog installed. There are also several subdirectories with EXAMPLES, data FILES, and TEST files.

- Remake the executables with the provided shell script ***mkngac***. After doing so you should be able to execute ***ngac*** directly from a command shell prompt. If you do this skip through the following steps to "Now you should see …" below.
- Otherwise, in the source directory start SWI-Prolog from a command shell prompt using the name of the SWI-Prolog executable (usually 'swipl', 'swi-pl', or something similar, depending on how it was installed).
- After printing a short banner SWI-Prolog will display its prompt "?- ".
- At the Prolog prompt enter "[ngac]." (not the quotes, but do enter the full stop) The top level file ***ngac.pl*** ensures that all the other necessary modules are loaded.
- Prolog will compile the code and print "true."
- Execute the code by entering at the Prolog prompt "ngac." This is the entry point for the Policy Tool.
- Now you should see the 'ngac' prompt "ngac>"
- The 'ngac' commands, documented in D2.2 section 5.3.2 as well as in a release note in the source directory, are now available. Entering the command "help" will list the available commands in the current mode (there are two modes: admin and advanced, the latter used primarily for development and testing).

- **Usage Guide**

The 'ngac' policy tool presents a command line interface that offers a defined set of commands. The command interpreter also offers selective tracing of commands for development and debugging of the tool itself. The 'ngac' command interpreter is easily extensible for new commands, and this ability has been frequently used during the implementation of the 'ngac' software. The syntax and simple semantic checking of commands are achieved declaratively, and the addition of a new command is straightforward. There are two levels of commands: a restricted set for ordinary users (typically security administrators) and an expanded set that includes commands that are primarily of use to the tool developers.

The 'ngac' tool has some self-tests built in. These may be run to ensure that everything is working correctly after installation, or after source code changes are made. The self-tests can be run by starting 'ngac' normally and entering at the 'ngac>' prompt the command "selftest."

There are several example policies included with the sources of the ngac Policy Tool. These include examples from the NGAC literature and are described in other project documents.

There are predefined command procedures (“procs”) that run some examples. At the ngac> prompt a predefined procedure (e.g. one may develop a command procedure named “myproc”) can be run with the command `proc(myproc).` It can be run with verbose output with the command `proc(myproc,verbose).` It can be run with a pause before each command (useful for demos) with the command `proc(myproc,step).`

It is instructive to read the file ***procs.pl*** that defines the predefined procedures. The procedures each consists of a few of the commands available at the ngac> prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

‘ngac’ commands can also be put into a file and executed as a script, without modifying the ***procs.pl*** file, using the `script` command, which like `proc` accepts the optional arguments `verbose` and `step`.

- **‘ngac-server’ Deployment Procedure**

The ngac server should be started with a compiled executable. This is preferable since it allows the command line options to be specified. Consult deliverable D1.4 or the release note for the command options.

There is a shell script of curl commands included with the source (servercurltest.sh) in the ***TEST*** subdirectory. This script can be run to send a sequence of requests to the server to test for known correct answers. (This script does not run a complete set of tests, it’s just intended to test that the server is responding.)

- **Manual usage**

If you do want to start the server from the policy tool follow the instructions above to get ‘ngac’ running. After starting ‘ngac’ it offers the prompt “ngac>”. There are a set of basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Entering the command “help” will list the available commands in the current mode.

If you want to load any policy files, do it now with the ‘ngac’ command `import(policy(PolicyFileName)).`, where PolicyFileName is the name of a .pl file relative to the execution directory. You can also combine policies with the ‘ngac’ `compose` command. When you have the desired policies loaded and composed, start the server from the ‘ngac’ tool using the command `server(PortNumber).`, where PortNumber is an unused TCP port. The server will be started and will be listening to that port for calls to its RESTful API. A server started in this way will expect the default admin token (the string “admin_token”, without the quotes) in the policy administration API calls. Starting the server executable file provides the opportunity to set an alternative admin token. (It is also possible to change the admin token from the 'ngac' command line using the `set` command, before issuing the `server` command.)

- **Background usage**

The 'ngac-server' can be run in the background from a startup script. To the command 'ngac-server' add any desired command line options and append a "&" to run the process in the background.

- **Configuration Guide**

The tool can be easily extended in several ways.

- Commands can be added by modifying the `command` module to add a `syntax`, `semantics` (optional), `help`, and `do` clause for the new command. A `syntax` clause must be added for the command. This clause declares the command name and parameters, and what mode the command belongs to, admin or advanced. Admin commands are available in admin mode, but also accessible in the advanced mode but not vice versa.

- The self-test framework is implemented in the `test` module. Tests for specific new modules can be added in the `TEST` subdirectory. An example of a test definition file for the `spld` module is implemented in `TEST/spld_test.pl`.

- New predefined 'ngac' command procedures can be added to the `procs` module. A `proc` clause is added for each new procedure to be defined. There are examples in the `procs.pl` file.

Global parameters are set in the `param` module. Settable parameters (those that can be changed from the 'ngac' command line with the `set` command) are itemized in a list `settable_params`. Adding new settable parameters requires the new parameter name to be added to this list and to the `dynamic` directive above it in a fashion similar to the other entries.

4.12 MODEL-BASED TESTING (MBT)

MBT is designed and implemented as four different activities in PHANTOM and thus the deployment guide contains the following guide.

In addition to working installation of TITAN and DIVERSITY that you can obtain from the Eclipse foundation platform:

4.12.1 Model Simulation

- **System requirements**

The model simulation is based on Windows OS, under the condition that the dependencies listed below is correctly installed with the proper configuration.

- **Dependencies**

The model simulation is achieved by DIVERSITY tool [7], so the dependency is to install DIVERSITY in Windows.

DIVERSITY has an issue with TTCN-3 publication and you will need this additional scripts to correct DIVERSITY output

https://github.com/PHANTOM-Platform/MBT-TTCN3-Publisher-for-Diversity

- **Deployment Procedure**

The deployment procedure is as follows:

1. Download the installation package from [7] to any directory in the OS.

2. Run the avm.exe file to install the tool

- **Usage Guide**

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. The user is expected to develop a model to simulate or import an existing model to the DIVERSITY tool, and then the tool will start the simulation process and output results.

4.12.2 Performance Estimation

- **System requirements**

The MBT system is generally independent of hardware and can run upon different hardware with Linux or Windows OS, under the condition that the dependencies listed below is correctly installed with the proper configuration.

- **Dependencies**

The Performance Estimation is developed in the languages of Python (version 3), so the dependency to run MBT system is to have python compiler installed in the OS.

Scripts are available on github:

https://github.com/easy-global-market/PHANTOM_Performance_Estimator

- **Deployment Procedure**

The deployment procedure is described as follows:

1. Unzip the source to any directory in the OS.

2. Run the performance estimator via the following command, where [input_component_network] is the input component network with full name and path.

```
python PerformanceEstimator.py [input_component_network]
```

- **Configuration Guide**

The following two variables can be configured at the beginning of the source code in PerformanceEstimator.py file:

```
tested_component_results_file_path
output_estimation_result_file_path
```

The "tested_component_results_file_path" indicates where the previous testing result file can be found to support the performance estimation, and the "output_estimation_result_file_path" indiciates the target file to store the performance estimation result.

- **Usage Guide**

Performance estimation estimates the non-functional properties (e.g., execution time, energy consumption) of newly designed applications by analyzing PHANTOM component network and previous non-functional testing results. The input is the xml file of the "ComponentNetwork" of an application describing the inner components and their communications following PHANTOM specification, while the output is the estimation result to terminal and destination xml file. No external framework is needed to support the execution.

The user defines a component network file and running the command above will start the estimation process and output result.

4.12.3 Test Execution for Functional Testing and Non-functional Testing

- **System requirements**

The MBT test execution system is generally independent of hardware and can run upon different hardware with Linux or Windows OS, under the condition that the dependencies listed below is correctly installed with the proper configuration.

- **Dependencies**

The MBT system is developed in the languages of TTCN-3 (for testing control part) and C++ (for execution part), so the dependency to run MBT system is to have both C++ and TTCN-3 compiler installed in the OS.

C++ compiler: Any of the mainstream C++ compilers (e.g., GCC, MinGW, etc.) are enough.

TTCN-3 compiler: Eclipse Titan [8] is a TTCN-3 compilation and execution support with an Eclipse-based IDE. Two ways exist to install Titan in the system, the first way is to directly download and install the precompiled version of Titan from [9], and the second is to download the source code from [10] and compile it from sources.

Scripts to compile and execute the PHANTOM testcases are available on github:

https://github.com/PHANTOM-Platform/MBT-Test-Execution

- **Deployment Procedure**

The deployment procedure is described as follows:

1. Unzip the source to any directory in the OS.

2. run the nstall.sh from mbttest directory, by /install.sh, this step will

- creates a ./bin directory

- creates symlinks to the necessary source files

- creates a Makefile suitable to compile the project

- compiles the project having the final executable name to be "mbttest"

3. Edit the configuration file MyExample.cfg using any test editor to specify the server addresses, the inputs and output files locations, and the different project parameters.

4. Run the MBT systems by using the following command under the directory ./bin

- ttcn3_start ./mbttest ./MyExample.cfg

- **Configuration Guide**

The configuration of the MBT system is done by modifying the configuration file "MyExample.cfg" under the source file directory. The configuration file mainly contains the following information.

- Server and ports information of PHANTOM modules to interacts
- Authentication parameters for users/password/tokens
- File upload and download parameters to indicate the inputs files and output files location

- **Usage Guide**

Running the MBT system will execute the test cases specific to each use case application. Specifically, during the execution, the MBT system realizes the following steps to examine the functional and non-functional correctness of the use case.

Step 1. Testing system sends a) *testing inputs* (files), and b) *deployment plan* (xml file) to Repository;

*deployment plan indicates on which hardware the application components should be executed.

Step 2. Testing system sends the *execution request* (JSON message) to Application Manager, and receives the *execution id* (JSON message) from Application Manager;

*execution id is a unique identifier to identify one specific execution of an application.

Step 3. Testing system subscribes to the execution status by sending a *subscription request* (JSON message) to Execution Manager;

Step 4. Testing system waits until receives the notification (JSON message) from Execution Manager, indicating the execution is over;

Step 5. Testing system retrieves *execution outputs* (files) from Repository;

Step 6. Testing system retrieves *performance Information* (JSON message) from Execution Manager; (Get query)

Step 7. Testing system generates testing verdicts based on the real outputs and/or performance information.

4.13 APPLICATION MANAGER

This section describes the deployment of the PHANTOM Application Manager and provides a usage guide. These instructions may evolve fruit to extensions and improvements of the repository resulting from future work. Such updates will be available at [1]. Additionally, there are some video tutorials available at the PHANTOM YouTube channel [2].

The PHANTOM Application Manager server is composed of two components: a web server (http://) and a WebSocket server (ws://). The web server provides various functionalities for data query and data analysis via RESTful APIs with documents in JSON format. The server's URL is "http://localhost:8500" by default. The default port for the https service is 8510.

PHANTOM Application Manager server keeps track of the status of the running tasks (PHANTOM tools).

This server allows users to subscribe by using web sockets to get notifications on the changes on the tasks' status. Such notifications consists on forwarding a copy of the uploaded JSON.

4.13.1 Prerequisites

The server is implemented using Node.js, and connects to ElasticSearch to store and access Metadata. Before you start installing the required components, please note that the installation and setup steps mentioned below assume that you are running a current Linux as the operating system. The installation was tested with Ubuntu 16.04 and 17.04 LTS. Before you can proceed, please clone the repository:

```
git clone https://github.com/PHANTOM-Platform/Application-Manager.git;
```

4.13.2 Dependencies

This project has the following dependencies:

Component	Homepage	Version
Elasticsearch	https://www.elastic.co/products/elasticsearch	= 2.4.6
Node.js	https://apr.apache.org/	>= 4.5
npm	https://www.npmjs.com/	>=1.3.6

4.13.3 Installation

Before the installation of the Application Manager it is required that ***npm*** be installed in advance. Once it is successfully installed. The Repository and the other required packages are automatically installed running one of the following scripts. Please choose the appropriate shell scripts depending on your Operating System:

Shell script for Intel-x86 32bits (tested on Ubuntu):

```
bash setup-server-x86-32.sh
```

or the Shell script for Intel-x86 64bits (tested on Ubuntu):

```
bash setup-server-x86-64.sh
```

or the Shell script for Armv7l 64bits (tested on Raspbian):

```
bash setup-server-armv7-64.sh
```

The default port is 8500, which can be modified at the file appmanager_app.js.

4.13.4 Start/Stop the server

Starting the PHANTOM Application Manager by ***executing*** the next script. For security reasons, the services may not start if they are requested from root.

```
bash start-appmanager.sh;
```

The server can be stop with the script:

```
bash stop- appmanager.sh;
```

In case of issues it can be verified that the Application Manager is running and has access to the database with the next scripts.

Test of the Nodejs Front-end running service:

```
curl http://localhost:8500;
```

Test if the Front-end has access to the ElasticSearch database Server.

```
curl -s http://localhost:8500/verify_es_connection;
```

```
To access to the Web-Interface, access with your internet browser to
http://localhost:8500/appmanager.html
```

4.13.5 Configuration of USERS' accounts

After the installation, and before users can use the Application Manager, it is needed to register the users.

The script setup-new-server.sh provides an automatic method for register multiple users. In particular, the script registers the list of users_ids and passwords from the file list_of_users.ini.

```
bash setup-new-server.sh
```

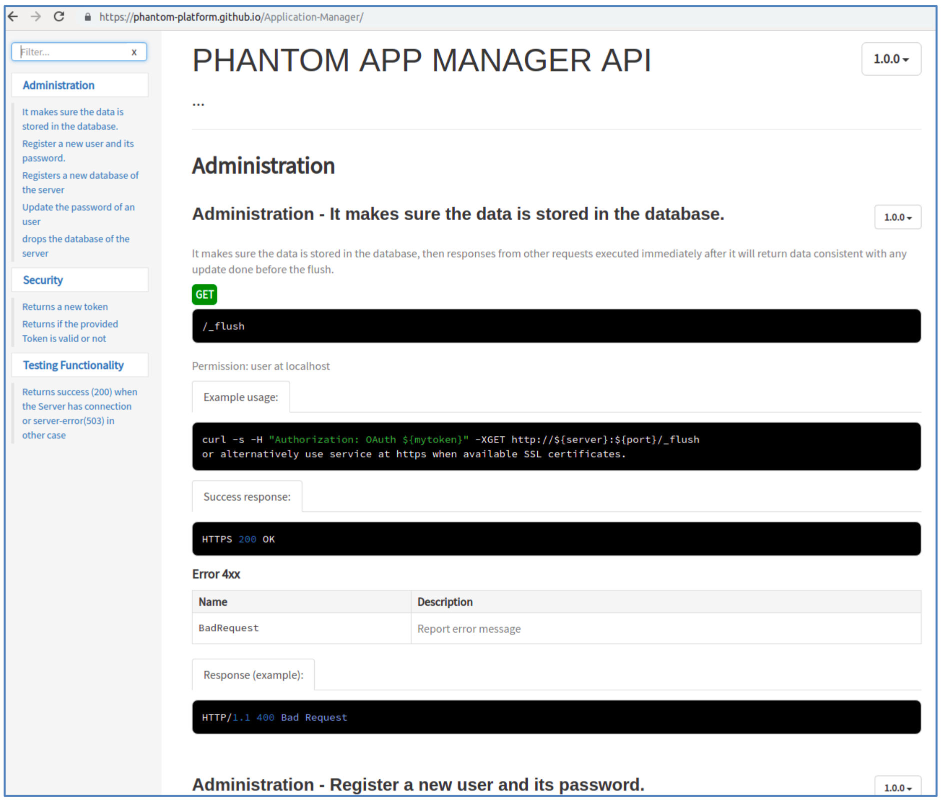
NOTICE: For security reasons, users' accounts can be ONLY registered on the server. Requests from different IPs will be rejected.

4.13.6 Example of use

The folders on the GitHub api_command_line, and api_bash_scripts show examples of using the PHANTOM Application Manager. Please access to those folders to get more details.

4.13.7 Reference manual of the RESTful API

The Reference manual of PHANTOM Application Manager RESTful API contains description of each of the implemented methods, with examples of use, and type of possible responses. The manual is built with the tool APIDOC, and formatted in a friendly HTML, where users can find a classification of the methods and a searching box. The manual is available online at https://phantom-platform.github.io/Application-Manager/ (see next figure).



https://phantom-platform.github.io/Application-Manager/

Filter... x

Administration
- It makes sure the data is stored in the database.
- Register a new user and its password.
- Registers a new database of the server
- Update the password of an user
- drops the database of the server

Security
- Returns a new token
- Returns if the provided Token is valid or not

Testing Functionality
- Returns success (200) when the Server has connection or server-error(503) in other case

PHANTOM APP MANAGER API 1.0.0

...

Administration

Administration - It makes sure the data is stored in the database. 1.0.0

It makes sure the data is stored in the database, then responses from other requests executed immediately after it will return data consistent with any update done before the flush.

GET

```
/_flush
```

Permission: user at localhost

Example usage:

```
curl -s -H "Authorization: OAuth ${mytoken}" -XGET http://${server}:${port}/_flush
or alternatively use service at https when available SSL certificates.
```

Success response:

```
HTTPS 200 OK
```

Error 4xx

Name	Description
BadRequest	Report error message

Response (example):

```
HTTP/1.1 400 Bad Request
```

Administration - Register a new user and its password. 1.0.0

Figure 19. Screenshot of the Online Reference Manual of the Application Manager API

4.14 RESOURCE MANAGER

Server which keeps track of the Status of the Available hardware in the system.

This section describes the deployment of the PHANTOM Resource Manager and provides usage guides. These instructions may evolve fruit to extensions and improvements of the repository resulting from future work. Such updates will be available at [1]. Additionally, there are some video tutorials available at the PHANTOM YouTube channel [2].

The PHANTOM Resource-Manager server is composed of two components: a web server (http://) and a WebSocket server (ws://). The web server provides various functionalities for data query and data analysis via RESTful APIs with documents in JSON format. The server's URL is "http://localhost:8600" by default. The default port for the https service is 8610.

PHANTOM Resource Manager server keeps track of the availability of the hardware. This server allows users to subscribe by using web sockets to get notifications on the changes on the tasks' status. Such notifications consists on forwarding a copy of the uploaded JSON.

4.14.1 Prerequisites

The server is implemented using Node.js, and connects to ElasticSearch to store and access Metadata. Before you start installing the required components, please note that the installation and setup steps mentioned below assume that you are running a current Linux as the operating system. The installation was tested with Ubuntu 16.04 and 17.04 LTS. Before you can proceed, please clone the repository:

```
git clone https://github.com/PHANTOM-Platform/Resource-Manager.git;
```

4.14.2 Dependencies

This project has the following dependencies:

Component	Homepage	Version
Elasticsearch	https://www.elastic.co/products/elasticsearch	= 2.4.6
Node.js	https://apr.apache.org/	>= 4.5
npm	https://www.npmjs.com/	>=1.3.6

4.14.3 Installation

Before the installation of the Resource Manager it is required that ***npm*** be installed in advance. Once it is successfully installed. The Repository and the other required packages are automatically installed running one of the following scripts. Please choose the appropriate shell scripts depending on your Operating System:

Shell script for Intel-x86 32bits (tested on Ubuntu):

```
bash setup-server-x86-32.sh
```

or the Shell script for Intel-x86 64bits (tested on Ubuntu):

```
bash setup-server-x86-64.sh
```

or the Shell script for Armv7l 64bits (tested on Raspbian):

```
bash setup-server-armv7-64.sh
```

4.14.4 Start/Stop the server

Starting the PHANTOM Resource Manager by ***executing*** the next script. For security reasons, the services may not start if they are requested from root.

```
bash start-resomanager.sh;
```

The server can be stop with the script:

```
bash stop-resomanager.sh;
```

In case of issues it can be verified that the Resource Manager is running and has access to the database with the next scripts.

Test of the Nodejs Front-end running service:

```
curl http://localhost:8600;
```

Test if the Front-end has access to the ElasticSearch database Server.

```
curl -s http://localhost:8600/verify_es_connection;
```

```
To access to the Web-Interface, access with your internet browser to
http://localhost:8600/resourcemanager.html
```

4.14.5 Configuration of USERS' accounts

After the installation, and before users can use the Resource Manager, it is needed to register the users.

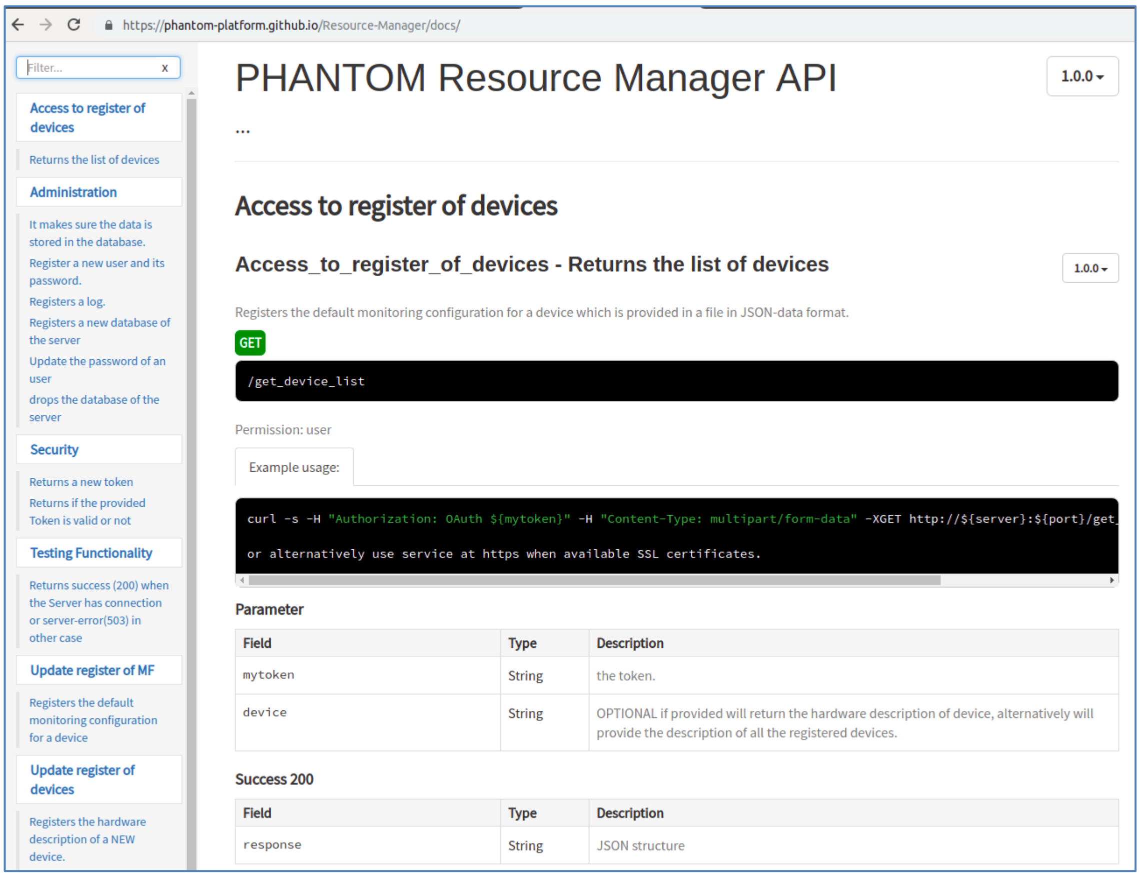
The script setup-new-server.sh provides an automatic method for register multiple users. In particular, the script registers the list of users_ids and passwords from the file list_of_users.ini.

```
bash setup-new-server.sh
```

NOTICE: For security reasons, users' accounts can be ONLY registered on the server. Requests from different IPs will be rejected.

4.14.6 Reference manual of the RESTful API

The Reference manual of PHANTOM Resource Manager RESTful API contains description of each of the implemented methods, with examples of use, and type of possible responses. The manual is built with the tool APIDOC, and formatted in a friendly HTML, where users can find a classification of the methods and a searching box. The manual is available online at https://phantom-platform.github.io/Resource-Manager/docs/ (see next figure).



https://phantom-platform.github.io/Resource-Manager/docs/

Filter... x

Access to register of devices
Returns the list of devices
Administration
It makes sure the data is stored in the database.
Register a new user and its password.
Registers a log.
Registers a new database of the server
Update the password of an user
drops the database of the server
Security
Returns a new token
Returns if the provided Token is valid or not
Testing Functionality
Returns success (200) when the Server has connection or server-error(503) in other case
Update register of MF
Registers the default monitoring configuration for a device
Update register of devices
Registers the hardware description of a NEW device.

PHANTOM Resource Manager API

1.0.0

...

Access to register of devices

Access_to_register_of_devices - Returns the list of devices

1.0.0

Registers the default monitoring configuration for a device which is provided in a file in JSON-data format.

GET

```
/get_device_list
```

Permission: user

Example usage:

```
curl -s -H "Authorization: OAuth ${mytoken}" -H "Content-Type: multipart/form-data" -XGET http://${server}:${port}/get_

or alternatively use service at https when available SSL certificates.
```

Parameter

Field	Type	Description
mytoken	String	the token.
device	String	OPTIONAL if provided will return the hardware description of device, alternatively will provide the description of all the registered devices.

Success 200

Field	Type	Description
response	String	JSON structure

Figure 20. Screenshot of the Online Reference Manual of the Resource Manager API

4.15 EXECUTION MANAGER

Server which keeps track of the Status of the Users' APPs, and historical brief reports of previous execution of applications.

4.15.1 Introduction

The PHANTOM Execution Manager server is composed of two components: a web server and a data storage system. The web server provides various functionalities for data query and data analysis via RESTful APIs with documents in JSON format. The server's URL is "http://localhost:8700" by default. The default port for the https service is 8710.

The Execution Manager process the collected data by the Monitoring-Server.

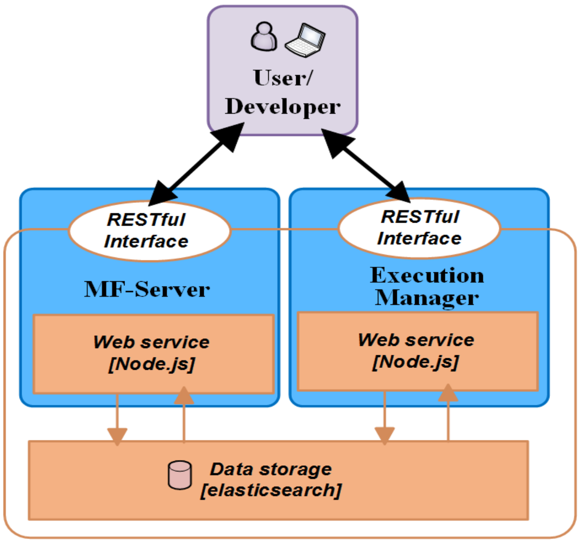


Figure 21: Execution Manager

4.15.2 Prerequisites

The server is implemented using Node.js, and connects to ElasticSearch to store and access Metadata. Before you start installing the required components, please note that the installation and setup steps mentioned below assume that you are running a current Linux as the operating system. The installation was tested with Ubuntu 16.04 LTS. Before you can proceed, please clone the repository:

```
git clone https://github.com/PHANTOM-Platform/Execution-Manager.git;
```

4.15.3 Dependencies

This project requires the following dependencies to be installed:

Component	Homepage	Version
Monitoring-Server	https://github.com/PHANTOM-Platform/Monitoring	
Elasticsearch	https://www.elastic.co/products/elasticsearch	= 2.4.6
Node.js	https://apr.apache.org/	>= 4.5
npm	https://www.npmjs.com/	>= 1.3.6

4.15.4 Installation

Before the installation of the Execution Manager it is required that ***npm*** be installed in advance. Once it is successfully installed. The Repository and the other required packages are automatically installed running one of the following scripts. The user can choose the appropriate shell scripts depending on their Operating System:

Shell script for Intel-x86 32bits (tested on Ubuntu):

```
bash setup-server-x86-32.sh
```

or the Shell script for Intel-x86 64bits (tested on Ubuntu):

```
bash setup-server-x86-64.sh
```

or the Shell script for Armv7l 64bits (tested on Raspbian):

```
bash setup-server-armv7-64.sh
```

The default port is 8700, which can be modified at the file repo_app.js.

4.15.5 Start/Stop the server

Starting the PHANTOM Exec Manager by ***executing*** the next script. For security reasons, the services may not start if they are requested from root.

```
bash start-execmanager.sh;
```

The server can be stop with the script:

```
bash stop-execmanager.sh;
```

In case of issues it can be verified that the Execution Manager is running and has access to the database with the next scripts.

Test of the Nodejs Front-end running service:

```
curl http://localhost:8700;
```

Test if the Front-end has access to the ElasticSearch database Server.

```
curl -s http://localhost:8700/verify_es_connection;
```

To access to the Web-Interface, access with your internet browser to http://localhost:8700/executionmanager.html

4.15.6 Configuration of USERS' accounts

After the installation, and user registration is required in order to use the Execution Manager.

The script setup-new-server.sh provides an automatic method for register multiple users. In particular, the script registers the list of users_ids and passwords from the file list_of_users.ini.

```
bash setup-new-execmanager-server.sh
```

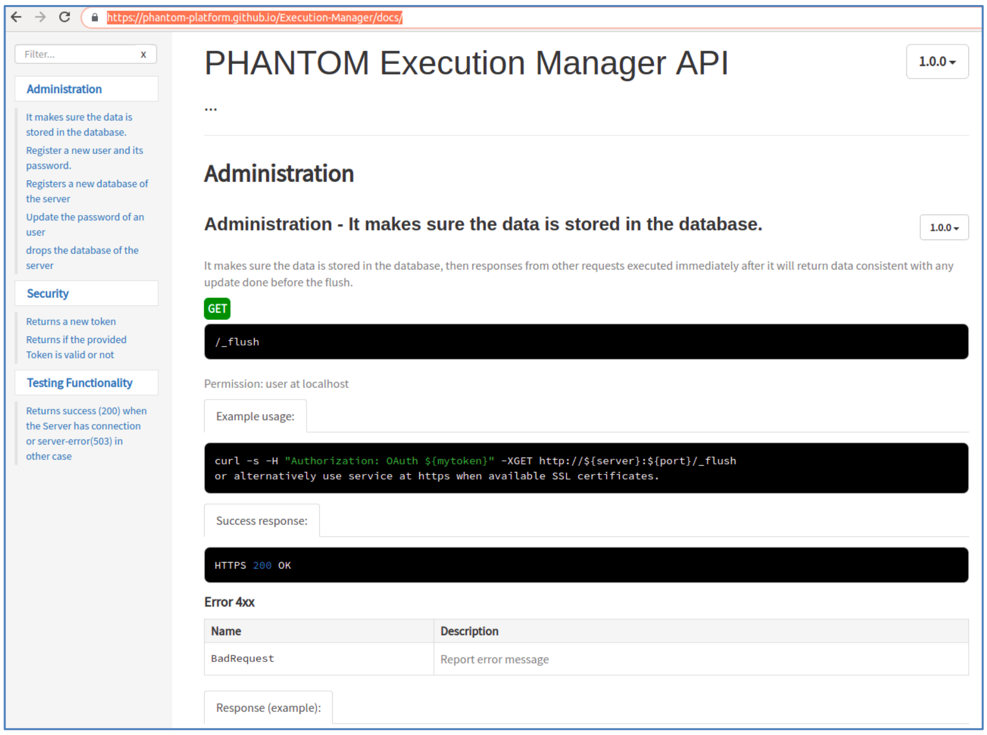
NOTICE: For security reasons, users' accounts can be ONLY registered on the server. Requests from different IPs will be rejected.

4.15.7 Example of use

The folder scripts shows examples of using the PHANTOM Execution-Manager. Please access to those folders to get more details.

4.15.8 Reference manual of the RESTful API

The Reference manual of PHANTOM Execution Manager RESTful API contains description of each of the implemented methods, with examples of use, and type of possible responses. The manual is built with the tool APIDOC, and formatted in a friendly HTML, where users can find a classification of the methods and a searching box. The manual is available online at https://phantom-platform.github.io/Execution-Manager/docs/ (see next figure).


https://phantom-platform.github.io/Execution-Manager/docs/
Filter...
Administration
It makes sure the data is stored in the database.
Register a new user and its password.
Registers a new database of the server
Update the password of an user
drops the database of the server
Security
Returns a new token
Returns if the provided Token is valid or not
Testing Functionality
Returns success (200) when the Server has connection or server-error(503) in other case
1.0.0
GET
Example usage:
Success response:
Response (example):

PHANTOM Execution Manager API

...

Administration

Administration - It makes sure the data is stored in the database.

It makes sure the data is stored in the database, then responses from other requests executed immediately after it will return data consistent with any update done before the flush.

```
/_flush
```

Permission: user at localhost

```
curl -s -H "Authorization: OAuth ${mytoken}" -XGET http://${server}:${port}/_flush
or alternatively use service at https when available SSL certificates.
```

```
HTTPS 200 OK
```

Error 4xx

Name	Description
BadRequest	Report error message

Figure 22. Screenshot of the Online Reference Manual of the Execution Manager API

4.16 INTEGRATED REFERENCE SYSTEM USER SCRIPTS

The User-Scripts consist in a set of Python and Bash scripts that helps the user on the management of the PHANTOM tools deployed on virtual environment of the Integrated Reference System, and on the automatization of the processing of the application using PHANTOM tool flow.

4.16.1 System requirements

The User Scripts depend on the Bash scripts developed for Linux environment, the current mechanism to launch PHANTOM tools is based on 'Xterm' and their behaviour was only tested on Linux (Ubuntu 16.04) environment.

To able run some PHANTOM tools they need to be deployed in the same machine:

- Parallelization Toolset;

- Generic Multi-Objective Mapper and Offline MOM;
- Deployment Manager; and
- Model-Based Testing tools

4.16.2 Dependencies

- Git
- Python3
- Python3 module: websocket-client
- Python3 module: Paramiko
- Xterm

4.16.3 Deployment Procedure

The deployment of the User-Scripts can be made using the git clone mechanism:

```
$git clone https://github.com/PHANTOM-Platform/PHANTOM-User-Scripts.git
```

This command will download all the scripts

4.16.4 Usage Guide

- **Configuration of 'settings.py' –** The file 'settings.py' must be configured to according to the characteristics of the application and the intentions of the USER. The 'settings.py' is divide in 3 main zones:
  - **Repositories configurations** – used for the USER specify the location of the repositories to be used (localhost or remote location) and credentials. E.g.:

    ```
    # Set the Repository IP address and port
    #repository_ip = "141.58.0.8"
    #repository_port = 2777
    repository_ip = "localhost"
    repository_port = 8000

    # Authentication credentials
    user =
    password =
    ```

  - **Application configurations** – Used for the USER to indentify application specific properties:

    Name of the application to be used server and by PHANTOM tools to identify the application:

    ```
    app_name = "WINGStest3"
    ```

Path for the root of the application's folder (to upload makefile and cla.in)
root_path = "/home/demo/phantom-tools/Examples/WINGStest3"

Path for the folder with the source code
src_path = "/home/demo/phantom-tools/Examples/WINGStest3/src"

Path for the folder with description files (Component Network and Platform Description)
desc_path = "/home/demo/phantom-tools/Examples/WINGStest3/description"

Path for thevfolder with PHANTOM API files
phantom_path = "/home/demo/Desktop/phantom-tools/PHANTOM_FILES"

Link to the where the marketplace is hosted and name of the folder where IPCores shoul be stored locally
ipMarket_path = "https://github.com/PHANTOM-Platform/PHANTOM-IP-Core-Marketplace.git"
ip_folder = "IPCore-MarketPlace"

Path for folder with application inputs
inputs_path = ""

Name of the component network file to be used
CompNetName = "cpn.xml"

Name of the platform description file to be used
PlatDesName = "hw_local.xml"

- **Tools Configurations** – This section contains the parameters for configuring the PHANTOM tools. In includes the path for each tool deployed on the machine. E.g.:

  # MOM location
  MOM_path = "/home/demo/phantom-tools/GenericMOM"

  And tool specific arguments. E.g.:

  PT_mode = "on" #operation mode: on | off - "on" to run PT normally, "off" to skip code analysis process

  In this section can also be found the property for the address of the FPGA VM, as well as the SSH port to be used

  FPGAVM_ip = ""
  FPGAVM_port =

- **Run the start script –** The last step consists in the execution of script that will: upload all the needed files to the specified repositories; register the application on the Application manager; and configure and launch each tool. To start this script run:

```
demo@ubuntu:~/phantom-tools/User-tools$./start-PHANTOM.py
```

This command as several options as shown the when using the '-h' flag:

```
demo@ubuntu:~/Desktop/phantom-tools/User-tools$ ./start-PHANTOM.py -h
usage: start-PHANTOM.py [-h] [-u] [-d] [-i] [-c] [-m] [-p]

Tool to support the execution of an application on PHANTOM Framework

optional arguments:
  -h, --help          show this help message and exit
  -u, --noUpload      Do not (re)upload the application to the repository.
                      (Application should be already in repository)
  -d, --onlyDesc      Only re-uploads the description files to the repository)
  -i, --skipInputs    Do not (re)upload the application inputs to the
                      repository. (Inputs should be already in repository)
  -c, --clean         Clean all the data in repositories and temporary cache
                      on PHANTOM tools. Automatically update PHANTOM_FILES
                      (-p)
  -m, --ipmarket      Uploads the IP Core Market place to the repository
  -p, --phantomfiles  Uploads the PHANTOM files (PHANTOM API and Monitoring
                      API)
```

5. CONCLUSION

This document depicts the final deployment configurations that are needed for deploying the PHANTOM tools/servers, as well as details about the applications' placement in the framework and its compliance to the PHANTOM Programming Model for its analysis and deployment, as they were formed during the latest integration activities. The instructions in this guide, can assure the reader that, if followed correctly, the tools will be successfully installed, configured and deployed on any machine(s) that satisfy the dependencies that are described for each individual tool.

6. References

[1] Source code available at https://github.com/PHANTOM-Platform.

[2] Demonstrations and tutorials available at https://www.youtube.com/channel/UCtl2wQYh_Nj3HbyFoM1XHqQ

[3] G. Clinton and T. Zachary, ElasticSearch: The Definitive Guide, O'Reilly, 2015.

[4] R. Kuc and M. Rogozinski, ElasticSearch Server, second edition, PACKT publishing, 2014.

[5] B. Dixit, ElasticSearch Essentials Paperback, 2016.

[6] A. Paro, ElasticSearch Cookbook , PACKT publishing, 2013

[7] Eclipse Modeling Project, https://projects.eclipse.org/projects/modeling

[8] Eclipse Titan, https://projects.eclipse.org/projects/tools.titan

[9] Eclipse Titan Download, https://projects.eclipse.org/projects/tools.titan/downloads

[10] eclipse_titan.core, https://github.com/eclipse/titan.core

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

A.1. Example Application

Component C0

```
#include "../phantom/phantom.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include "example_lib.h"

void *C0() {

       phantom_monitor *monitor = phantom_monitor_init();

       phantom_mf_start(monitor);

       #pragma phantom queue inQueue00
       phantom_queue *C0_Queue0 = phantom_queue_init("inQueue00");
       #pragma phantom signal inSignal0
       phantom_signal *signal = phantom_signal_init("inSignal0");
        uint8_t *a[3];
        uint32_t size;
        int i;
        for(i=0; i<3; i++) {
                if(i==0) size = 2;
                else if(i==1) size = 3;
                else if(i==2) size = 5;
                a[i] = (uint8_t
*)malloc(size*sizeof(uint8_t)+sizeof(uint32_t));
                memcpy(a[i],uint32_t_to_uint8_t(size),sizeof(uint32_t));
        }
        for(i=0; i<10; i++) {
                if(i<2) a[0][i+sizeof(uint32_t)] = i;
                else if(i<5) a[1][i-2+sizeof(uint32_t)] = i;
                else a[2][i-5+sizeof(uint32_t)] = i;
        }
        for(i=0; i<3; i++) {
       phantom_queue_put(C0_Queue0,a[i]);
        }

        phantom_wait(signal);

        uint8_t *d[3];
        for(i=0; i<3; i++) {
                d[i] = (uint8_t *)malloc(1+sizeof(uint32_t));
                memcpy(d[i],uint32_t_to_uint8_t(1),sizeof(uint32_t));
                d[i][sizeof(uint32_t)] = 20 + i;
             printf("Putting item: [%d] in
queue...\n\n",d[i][sizeof(uint32_t)]);
             phantom_queue_put(C0_Queue0,d[i]);
        }

       sleep(10);
       phantom_mf_end(monitor);
}
```

Component C1

```
#include "../phantom/phantom.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include "example_lib.h"
```

```
void *C1() {

	phantom_monitor *monitor = phantom_monitor_init();

	phantom_mf_start(monitor);

	#pragma phantom queue outQueue01
	phantom_queue *C1_Queue0 = phantom_queue_init("outQueue01");

	phantom_shared *shared_object = phantom_shared_init("outShared0");

	#pragma phantom signal inSignal1
	phantom_signal *signal1 = phantom_signal_init("inSignal1");

	#pragma phantom signal outSignal2
	phantom_signal *signal2 = phantom_signal_init("outSignal2");

	uint8_t *a[6];

	  int i,j;
	  uint32_t size;
	  for(i=0; i<6; i++) {
a[i] = (uint8_t *)phantom_queue_get(C1_Queue0);
			size = uint8_t_to_uint32_t(a[i]);
			printf("Receiver: printing a[%d] --> ",i);
			for(j=0; j<size; j++)
				printf("%d ",a[i][j+sizeof(uint32_t)]);
			printf("\n");
	  }
	  phantom_notify(signal1);

	#pragma phantom shared outShared0
	int *shared_data1 = (int *)malloc(sizeof(int *) * 10);


	  printf("\nC1: Local data -> ");
	  for(i=0; i<10; i++) {
			shared_data1[i] = i;
			printf("%d ",shared_data1[i]);
	  }
	  printf("\n\n");

	  phantom_wait(signal2);
	  phantom_synchronize(shared_object,shared_data1,0);

	  printf("\nC1: Synced data -> ");
	  for(i=0; i<10; i++) {
			printf("%d ",shared_data1[i]);
	  }
	  printf("\n\n");
	sleep(5);
	phantom_mf_end(monitor);
}
```

Component C2

```
#include "../phantom/phantom.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include "example_lib.h"

void *C2() {
```

```
phantom_monitor *monitor = phantom_monitor_init();
phantom_mf_start(monitor);

#pragma phantom queue outQueue02
phantom_queue *C2_Queue0 = phantom_queue_init("outQueue02");

phantom_shared *shared_object = phantom_shared_init("inShared0");

#pragma phantom signal outSignal0
phantom_signal *signal1 = phantom_signal_init("outSignal0");

#pragma phantom signal outSignal1
phantom_signal *signal2 = phantom_signal_init("outSignal1");

#pragma phantom signal inSignal2
phantom_signal *signal3 = phantom_signal_init("inSignal2");

 uint8_t *a;
 uint8_t *b;
 uint8_t *c;
 a = (uint8_t *)malloc(2*sizeof(uint8_t)+sizeof(uint32_t));
 b = (uint8_t *)malloc(3*sizeof(uint8_t)+sizeof(uint32_t));
 c = (uint8_t *)malloc(5*sizeof(uint8_t)+sizeof(uint32_t));
 memcpy(a,uint32_t_to_uint8_t(2),sizeof(uint32_t));
 memcpy(b,uint32_t_to_uint8_t(3),sizeof(uint32_t));
 memcpy(c,uint32_t_to_uint8_t(5),sizeof(uint32_t));
 int i, j;
 for(i=0; i<10; i++) {
         if(i<2) a[i+sizeof(uint32_t)] = i+10;
         else if(i<5) b[i-2+sizeof(uint32_t)] = i+10;
         else c[i-5+sizeof(uint32_t)] = i+10;
 }
phantom_queue_put(C2_Queue0,a);
phantom_queue_put(C2_Queue0,b);
phantom_queue_put(C2_Queue0,c);

 phantom_notify(signal1);
 phantom_wait(signal2);
 uint8_t *d[3];
 uint32_t size;
 for(i=0; i<3; i++) {
       d[i] = (uint8_t *)phantom_queue_get(C2_Queue0);
         size = uint8_t_to_uint32_t(d[i]);
         printf("C2: printing d[%d] --> ",i);
         for(j=0; j<size; j++)
                 printf("%d ",d[i][j+sizeof(uint32_t)]);
         printf("\n");
 }

#pragma phantom shared inShared0
int *shared_data2 = (int *)malloc(sizeof(int *) * 10);


 printf("\nC2: Outgoing data -> ");
 for(i=0; i<10; i++) {
         shared_data2[i] = d[i%3][sizeof(uint32_t)];
         printf("%d ",shared_data2[i]);
 }
 printf("\n\n");

 phantom_synchronize(shared_object,shared_data2,1);
 phantom_notify(signal3);

sleep(7);
```

```
	phantom_mf_end(monitor);
}
```

A.2. Programming Model functions & annotations - final documentation

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

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

```
void *phantom_queue_get(phantom_queue *queue);
```
Returns and deletes the first item in the queue. Blocks until data is inserted in the queue.

```
bool phantom_queue_put(phantom_queue *queue,void *item);
```
Puts item at the end of the queue. Returns true if successful.

```
void *phantom_queue_peek(phantom_queue *queue);
```
Returns the first item in the queue. Returns NULL if queue is empty.

```
uint32_t phantom_queue_count(phantom_queue *queue);
```
Returns the number of items inside the queue.

Signal
A protocol used to coordinate the execution of components.

```
int phantom_notify(phantom_signal *signal);
```
Blocks the current thread/process until the signal in question is notified.

```
int phantom_wait(phantom_signal *signal);
```
Unblock a random single thread/process waiting on the signal.

```
void phantom_barrier(phantom_signal *signal);
```
A barrier function able to wait until all threads or processes, before that call have finished their work.

```
int phantom_notifyall(phantom_signal *signal);
```
Unblock all threads/processes waiting on the signal.

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

```
int phantom_lock(phantom_mutex *mutex);
```
Block until the current mutex can be owned by the requesting component.

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

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

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

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

```
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.

File Operations

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

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

- filename – is a C string describing items in the PHANTOM repository. Paths should be case sensitive, absolute paths, with the forward slash character as a path separator.
- mode is treated the same as mode in the POSIX fopen function.

int phantom_fclose (FILE * stream)

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

- stream - Pointer to a FILE object that specifies the stream to be closed.
- If the stream is successfully closed, a zero value is returned.
  On failure, EOF is returned.

int phantom_fflush (FILE * stream)

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

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

Writes an array of *count* elements, each one with a size of *size* bytes, from the block of memory pointed by *ptr* to the current position in the *stream*.

- ptr – Pointer to the input buffer
- size – Size in bytes of each element to write
- count – Number of elements to write
- stream – A pointer to a FILE object as returned from phantom_fopen
- The total number of elements successfully written is returned. Sets ferror if an error occurred.

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

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

- ptr – Pointer to the output buffer
- size – Size in bytes of each element to read
- count – Number of elements to read
- stream – A pointer to a FILE object as returned from phantom_fopen
- The total number of elements successfully read is returned.

int phantom_fgetpos (FILE * stream, fpos_t * pos)

Retrieves the current position in the *stream*.

- stream – Pointer to the stream object.
- pos – An allocated fpos_t which is filled with the current position.
- Returns zero on success or a platform-specific error number.

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

Sets the position indicator associated with the *stream* to a new position. If the stream is binary, the new position is offset + origin. If the stream is text, then offset shall be 0.

Returns zero on success or a platform-specific error number.

int phantom_fileno(FILE *stream)
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 may not be meaningful but can still be used to restore the position to the same position later using *phantom_fseek*.

Optimization Annotations

#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.

Serialization interfaces

Interface for serialization and deserialization of simple data types, strings, arrays, C++ vectors, and C++ vector of vectors.

- Simple data types

  ```
  size_t serialize_val(type in, char *out);

  size_t deserialize_val(type *out, char *in);
  ```

  type must be one of the following:
  - size_t
  - int8_t
  - int32_t
  - uint32_t
  - double
  - float
  - bool

  in is the serialization/deserialization input buffer.
  out is the output buffer.

- Strings

  ```
  size_t serialize_str(type in, char *out);
  ```

  type must be one of the following:
  - string
  - const int8_t*

  in is the serialization input buffer.
  out is the output buffer.

  ```
  size_t deserialize_str(type *out, char *in);
  ```

  type must be one of the following:
  - string
  - const int8_t

  in is the deserialization input buffer.
  out is the output buffer.

- C arrays

  ```
  size_t serialize_arr(type *arr, size_t size, char *out);

  size_t deserialize_arr(type *arr, size_t size, char *in);
  ```

  type must be one of the following:
  - const int8_t
  - double

- int32_t

arr is the serialization/deserialization input/output buffer.
size is the size of *arr*
in is the deserialization input buffer.
out is the output buffer

- C++ vectors

```
size_t serialize_vec(vector<type> *vec, char *out, bool w_size);

size_t deserialize_vec(vector<type> *vec, char *in, bool r_size);
```

type must be one of the following:

- bool
- double
- float
- int32_t

w_size is used to indicate if the size of the vector must be prepended to *out*.
r_size is used to indicate if the size of the vector must be read from *in*.
vec is the serialization/deserialization input/output buffer.
in is the deserialization input buffer.
out is the output buffer

- Matrices
  - C++ vector of vectors

```
size_t  serialize_mat(vector<vector<type> >  *mat,  char  *out,
bool w_size);

size_t  deserialize_mat(vector<vector<type> >  *mat,  char  *in,
bool r_size);
```

type must be one of the following:

- bool
- double
- float
- int32_t

w_size is used to indicate if the size of the vector must be prepended to *out*.
r_size is used to indicate if the size of the vector must be read from *in*.
mat is the serialization/deserialization input/output buffer.
in is the deserialization input buffer.
out is the output buffer.

  - C array of arrays

```
size_t  serialize_mat(type  **mat,  size_t  dim1,  size_t  dim2,
char *out);

size_t deserialize_mat(type **mat, size_t dim1, size_t dim2,
char *in);
```

type must be one of the following:

- const int8_t

dim1 and *dim2* correspond to the size of the first and second dimension of the matrix.
mat is the serialization/deserialization input/output buffer.
in is the deserialization input buffer.
out is the output buffer.

- 3D Matrices

```
size_t  serialize_3dMat(vector<vector<vector<type>  >  >  *mat,  char
*out, bool w_size);

size_t  deserialize_3dMat(vector<vector<vector<type>  >  >  *mat,  char
*in, bool r_size);
```

type must be one of the following:

- int32_t

w_size is used to indicate if the size of the vector must be prepended to *out*.
r_size is used to indicate if the size of the vector must be read from *in*.
mat is the serialization/deserialization input/output buffer.
in is the deserialization input buffer.
out is the output buffer.