# Training Material

Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models in the Cloud

H2020-ICT-2018-2020
Leadership in Enabling and Industrial Technologies: Information and Communication Technologies

Grant Agreement Number
871643

Duration
1 January 2020 –
30 June 2023

Deliverable reference
D7.4

Date
1 April 2023

Responsible partner
UPRC

Editor(s)
Alexandros Raikos,

Reviewers
Robert Gdowski, Amina Moussaoui

Distribution
Public

Availability
morphemic.cloud

Executive summary

This document presents the training materials for the MORPHEMIC training programmes in the context of the project's outreach and outlines the training activities roadmap which will be realised during the course of the MORPHEMIC project.

The training materials aim to deliver a clear and complete understanding of the platform's technical concepts, regular user interactions and underlying technologies. The training materials defined in this document were authored by the consortium partners responsible for their respective technical and knowledge contributions to the project.

Author(s)

Alexandros Raikos, Jean-Didier Totow Tom-ata, Efterpi Paraskevoulakou, Marta Różańska, Kyriakos Kritikos, Andreas Tsagkaropoulos, Joanna Chmielewska

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643

## Table of Contents

## Index of figures

## Index of tables

# 1 Introduction

This document presents the training materials which will be used in the MORPHEMIC project training programmes. The training materials, along with the training plan that will be defined in MORPHEMIC deliverable *D7.3 Training plan and activities*, are an integral part of the MORPHEMIC Work Package 7, Outreach, which aims to deliver a set of definitions, guidelines, and goals for carrying out training activities, as well as comprehensive technical guides and usage documentation. This deliverable also states the availability of the MORPHEMIC platform's training materials.

## 1.1 Deliverable Structure

Beginning in **chapter 2**, *Concerned Groups*, the basic trainee groups are defined, along with a brief description of the approach for each. **Chapter 3**, *Materials*, presents the training documentation and accompanying media which will be used in the training activities in detail. Furthermore, in **chapter 4**, *Availability*, an indicative collection of portals and other web-based repositories for publicly available MORPHEMIC training documentation will be presented. Finally, **chapter 5,** *Conclusions*, summarizes the contents of this deliverable.

## 1.2 Target Audience

The intended audience of this document consist of the MORPHEMIC project's consortium partners who are responsible for conducting activities described in the training plan. It is also directed to the project's consortium partners involved in outreach activities as part of the Work Package 7, Outreach, as defined in the project's Description of Action. Any other external parties interested in the definitions and the availability record of training materials of the MORPHEMIC project for the developed MORPHEMIC platform are included in this deliverable's target audience, as it is publicly accessible.

# 2 Concerned Groups

The MORPHEMIC training materials are a set of documents, videos, tutorials with the main objective of getting initiated with the MORPHEMIC platform and its technologies for a correct use. The MORPHEMIC training materials are addressing the following categories:

- **IT management**: The training materials describe all steps with details for installing and deploying MORPHEMIC platform. The training also contains examples for modelling cloud applications for different configurations supported by MORPHEMIC. The tools manipulating these models are presented in a manner to facilitate IT manager to have all required components for using and benefiting from all functionalities offered by MORPHEMIC platform.
- **DevOps**: DevOps engineer will find in the training materials information related to the application deployment on MORPHEMIC, monitoring and controlling different steps of application deployment process, tools for debugging, test and manipulating MORPHEMIC and running applications. Knowledge on the utility modelling is also provided.

# 3 Materials

The training materials presented in this chapter will equip the trainees with the necessary knowledge and skills to be able to effectively use and manage the MORPHEMIC platform, as well as understand the technical concepts and underlying technologies in use. The sub-sections below effectively divide the materials into two groups: IT management-related and DevOps-related. The former sub-section dedicated to IT management staff is focused more on the technical concepts and tools used to understand the platform's operation on a deeper technical context. The latter sub-section dedicated to DevOps trainees is focused on real world platform usage and context.

## 3.1 IT Management Materials

This section introduces the IT management training materials for the MORPHEMIC platform, where four main subsections make up this training program. Firstly, the CAMEL Modelling section details cloud application modelling using the CAMEL modelling language. Secondly, the MORPHEMIC Utility section provides an understanding of the utility approach in the modelling tools available on the platform to help optimize application deployments. Thirdly, the MORPHEMIC Monitoring section will cover the ways in which the MORPHEMIC platform monitors your IT infrastructure and identifies potential issues before they become critical. Lastly, the MORPHEMIC Adaptation Example section will provide you with a real-world example of how the platform can be used to adapt to changing business requirements.

### 3.1.1 CAMEL Modelling

CAMEL is a rich, multi-domain-specific language that enables to specify multi-cloud applications across multiple domains, which are related to the application lifecycle. CAMEL has been extended, from version 2.5 to 3.0, in the context of the MORPHEMIC project in order to support the modelling of polymorphic applications.

CAMEL has its own web page[1] which introduces CAMEL and provides pointers to various sources of information about CAMEL. Further, during the MELODIC project[2], multiple confluence pages were developed which covered detailed analysis of the modelling for all the domains involved in CAMEL. In addition, extra confluence pages were developed to: (a) clarify which CAMEL 2.5 features are supported by MELODIC platform, (b) provide some basic CAMEL 2.5 information in the form of a FAQ and (c) collect and supply all information sources related to CAMEL 2.5. All these pages were incorporated in a single page called Modelling[3].

As CAMEL 3.0 builds on CAMEL 2.5, the same structure was followed for producing the related material for CAMEL 3.0. In particular, a single landing page was created called Camel 3.0[4] in which we attached as sub-pages all the materials that we have developed. These sub-pages include:

- CAMEL 3.0 Information Sources[5] (see Section 5): this sub-page gathers all information sources related to CAMEL 3.0. The information sources are categorised as follows:
    - a documentation-based excel file that covers all CAMEL elements and their textual syntax,
    - a folder in CAMEL's Gitlab that provides multiple examples of CAMEL models,
    - confluence pages related to CAMEL. These are separated into two categories: (a) those related to CAMEL 2.5 and (b) those related to CAMEL 3.0,
    - editor related information sources covering mostly installation instructions as well as the wiki page of the graphical CAMEL editor, i.e., the CAMEL designer,
    - a pointer to CAMEL's meta-model and domain-specific source code,
    - a pointer to overall CAMEL project in Gitlab, including the source-code of the CAMEL textual editor.

[1] CAMEL Web Site
[2] MELODIC Project Web Site
[3] MELODIC's Confluence Page on Modelling (requires free registration)
[4] MORPHEMIC's Confluence Page on CAMEL 3.0 (requires free registration)
[5] MORPHEMIC's Confluence Page on CAMEL 3.0 Information Sources (requires free registration)

- CAMEL 2.5 vs. 3.0[6] (See Section 0): a summary of the extensions that have been made to CAMEL 2.5 in order to produce CAMEL 3.0.
- MORPHEMIC Support to CAMEL 3.0 features[7] (see Section 0): Here we explain which CAMEL 3.0 features are or will be supported by the MORPHEMIC platform. As CAMEL 3.0 builds on top of CAMEL 2.5, the analysis was split according to CAMEL 2.5 features and the extensions introduced by CAMEL 3.0. For CAMEL 2.5 features, the reader is forwarded to another confluence page[8] that explains the level of their support in terms of the MELODIC platform. While a table-like structure is supplied in order to explain the level of support to the new features of CAMEL (the so-called extensions).
- Attribute Modelling Extensions[9] (see Section a): here we analyse the extension done over attributes in order to express range-based constraints on feature characteristics like (number of cores in CPUs). The analysis is accompanied with various examples formulated in the CAMEL (3.0) textual syntax.
- Deployment Modelling Extensions[10] (see Section b): here we analyse the 4 main extensions made to the deployment domain: (a) reference of communications to their respective communication requirements, (b) association of component configurations to their own requirements set, (c) introduction of new configuration kinds and (d) introduction of control-flow relationships between application components. The sub-page is organised into 4 sections accordingly. In each section, apart from the analysis, we provide examples following the CAMEL (3.0) textual syntax.
- Requirements Modelling Extension[11] (see Section c): Here we analyse the extension related to introducing a new kind of hard requirement called communication requirement. The analysis is accompanied with a full example in CAMEL 3.0 syntax of a communication requirement (with constraints over network latency and jitter).
- Metric Modelling Extensions[12] (see Section d): in this sub-page we analyse in respective sections all extensions related to the metric domain, i.e.: (a) the ability to measure communication quality by referring to the respective communication (object) in object contexts, (b) the ability for metric variables to refer to metrics (such that they can be associated with and exploit the predictions related to these metrics in the context of deployment reasoning) and (c) window-related extensions which include the introduction of window processing and new kinds of window sizes. In all these sections the analysis is accompanied with examples in CAMEL 3.0 (textual) syntax.
- Metadata Modelling Extension[13] (see Section e): analysis of the extension made to the metadata domain in order to pinpoint which MetaData Schema (MDS) objects are implemented by the MORPHEMIC platform and can thus be utilised for annotating CAMEL models.
- CAMEL Configuration Alternatives[14] (see Section f): a guide towards utilising the different kinds of (application) component configurations that are supported by CAMEL 3.0.

Please note that along with the "Getting started with CAMEL 3.0" confluence page[15] (see Section 3.2.1) and this well-structured set of materials the modeller has all the knowledge required not only to get started with CAMEL but also master the CAMEL modelling thus being able to specify full-fledged CAMEL models for his/her application.

We acknowledge the fact that all this knowledge is part of a project-specific confluence web site (where registration is needed but is free). To this end, in order to provide also public access to such knowledge, the following actions will be performed:

- Publication of CAMEL's Getting started guide (see Section 3.2.1) in MORPHEMIC project web site.
- Gradual publication of all other CAMEL material from the project confluence in the CAMEL's web site.

[6] MORPHEMIC's Confluence Page Comparing CAMEL 2.5 and 3.0 (requires free registration)
[7] MORPHEMIC's Confluence Page on Supported CAMEL 3.0 Features (requires free registration)
[8] MELODIC's Confluence Page on Supported CAMEL 2.5 Features (requires free registration)
[9] MORPHEMIC's Confluence Page on CAMEL Attribute Modelling Extensions (requires free registration)
[10] MORPHEMIC's Confluence Page on CAMEL Deployment Modelling Extensions (requires free registration)
[11] MORPHEMIC's Confluence Page on CAMEL Requirements Modelling Extensions (requires free registration)
[12] MORPHEMIC's Confluence Page on CAMEL Metric Modelling Extensions (requires free registration)
[13] MORPHEMIC's Confluence Page on CAMEL Metadata Modelling Extensions (requires free registration)
[14] MORPHEMIC's Confluence Page on CAMEL Configuration Alternatives (requires free registration)
[15] MORPHEMIC's Confluence Page on CAMEL Getting Started Guide (requires free registration)

- Inclusion of all CAMEL 3.0 related material in this deliverable's Appendix A (see Section 5).

### 3.1.2 MORPHEMIC Utility

#### *3.1.2.1 Optimization problem*

Cloud applications can be considered as a set of communicating components, $\boldsymbol{C}$ , where the term component should be understood as a part of the application, which can be a software component, a container, a serverless function, platform provided software like databases or data lakes, or third-party external web servers. A persisted application must be able to respond to variation in workload and other changes in its environment happening during its extended execution time. This requires that there is some variability allowed for the configuration of the components. In general, a component, $C_i \in \boldsymbol{C}$ , has a set of quality attributes, $\boldsymbol{A}_i$, that allows the behaviour of the component to be adapted. Examples of an attribute, $a_{i,j} \in \boldsymbol{A}_i$, may be the number of cores given to the component, the encoding format of a video stream that can be changed to save power or bandwidth etc. Each attribute takes its value from its domain, $a_{i,j} \in \boldsymbol{V}_{i,j}$, which can be a continuous range or a discrete set of options. The domain is fixed since it represents the various configuration options for that particular attribute. The Cartesian product of the attribute domains represents the variability space of the component, i.e., the number of different variants that can be deployed, $\boldsymbol{V_i} = V_{i,1} \times V_{i,2} \times \ldots \times V_{i,|A_i|}$. Similarly, the variability space for the whole application will be the Cartesian product of the variability spaces for all its components, $\boldsymbol{V} = V_1 \times V_2 \times \ldots \times V_{|\boldsymbol{C}|}$. The values assigned to all application attributes at a particular time is called the application's configuration and can be ordered as a vector $\boldsymbol{c}(t) \in \boldsymbol{V}$.

The application must be monitored to detect when there is a need to reconfigure the application. The measurement events occur at discrete time points, $t_k$, and one cannot assume that the time points are equidistant, e.g. consider for instance the monitoring event representing the arrival or departure of a user of the application. The most recent measurements, the metrics, available at the time point can either depend on the application's configuration or be independent of the configuration. The number of active users is an example of the latter type, and the response time recorded for a user's request is an example of the former. The vector of the dependent metrics will be denoted $\boldsymbol{\theta}_D\left(t_k \,|\boldsymbol{c}(t_k)\right)$ where the vertical line indicates that the values are conditioned on the given configuration. $\boldsymbol{\theta}_I(t_k)$ will be used to indicate the vector of the independent metric values. The overall metric vector is just the concatenation of the two types of metrics $\boldsymbol{\theta}(t_k)$. The metric values may not be directly useful for the application management if they change too quickly as reconfigurations take time, and so they should better be filtered by various functions as performance indices denoted $\boldsymbol{\psi}\left(t_k \,|\, \boldsymbol{c}(t_k)\right)$. For instance, the response time measure for an individual user request may mostly indicate the complexity of that request whereas the windowed average of the most recent response times for all users says something about the application performance and the user experience. These functional combinations of the metric values together with the metric vector itself jointly form the application's execution context.

The application is deployed for a reason, and the application's owner has goals and preferences for the management of the application. These must be captured for autonomic and optimised operations, and the rational choice would be to optimise the owner's utility. The best way to capture utility is through a utility function mapping from the application's configuration to a goodness value in the unit interval. Hence, finding the optimal application configuration, $\boldsymbol{c}^*(t_k)$, implies solving the following mathematical programme for the application's execution context.

$$\boldsymbol{c}^*(t_k) = argmax_{c \in V} U\left(\boldsymbol{c} \,\middle|\, \boldsymbol{\theta}\,(t_k \,|\, \boldsymbol{c}), \boldsymbol{\psi}(t_k \,|\boldsymbol{c}\,)\right)$$

#### *3.1.2.2 Reasoning process*

Optimization of the application configuration is a complex task that is difficult, or even impossible to be performed by a human. There are many offers and possible application configurations, and therefore there is a need for a management tool that can work as a middleware and support the optimization process.

The application can be managed by the autonomic computing deployment platform MORPHEMIC. The reasoning process in MORPHEMIC is inspired by the MAPE-K loop[16], which stands for Monitoring, Analysing, Planning, and Executing under the gathered Knowledge.

The reasoning process can be seen on Figure 1. Firstly, there is a need to model the application with its sensors and metrics, and finally, the deployment goals: what is important for the application owner. Then during running of the application, it is being monitored and when the currently running configuration is no longer considered optimal, the step of analysis and planning takes place. This is when the decision-making process takes place. It considers all criteria important for the application owner like cost and performance. The optimization is performed with the usage of the utility function. When the best configuration is found, the next step is the execution of the plan, which means the deployment or redeployment of the application into the chosen Cloud provider and to the chosen configuration. When the application is re-deployed, it is still continuously monitored by the application sensors. This is how the application always runs on the most optimal configuration for the current context.

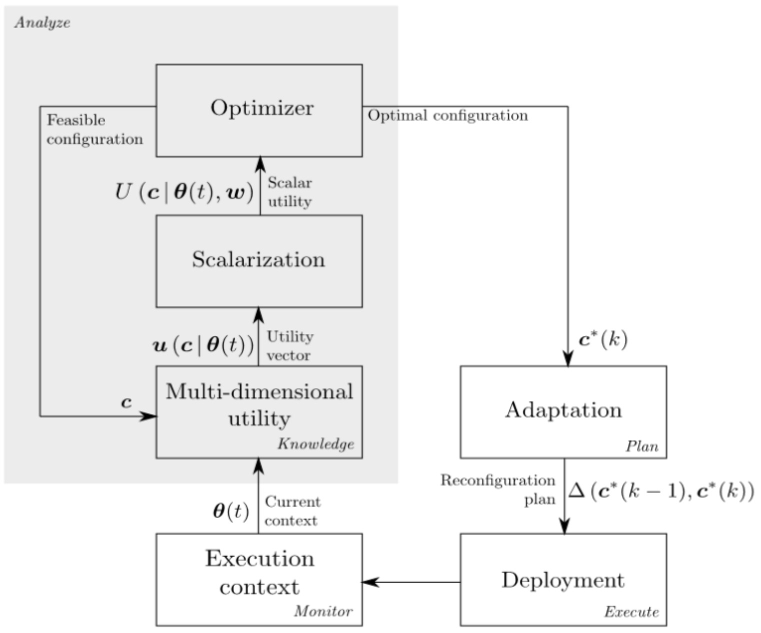

*Figure 1 The MORPHEMIC reasoning process*

The utility function modelling process can be learnt based on the example of Genome application that was created for MELODIC project and explained in detail in MORPHEMIC deliverable *D3.3 Optimized planning and adaptation approach*. The high-level goal of the adaptation is to add more workers if it is needed or delete some workers if they are not necessary to finish Genome calculations on time. The main performance indicator is the estimated remaining time.

1. The first step is to provide the sensors to calculate the number of trainings done, the time of the last training, and elapsed wall clock execution time. The details of these metrics are:
   A. Raw metric values when a job finishes:
      - Number of trainings done: $k$
      - Time of the last training: $\theta_5(k)$
      - Elapsed wall clock execution time: $\theta_3(k)$

[16] Romero-Garcés A, Hidalgo-Paniagua A, González-García M, Bandera A. On Managing Knowledge for MAPE-K Loops in Self-Adaptive Robotics Using a Graph-Based Runtime Model. *Applied Sciences*. 2022; 12(17):8583. https://doi.org/10.3390/app12178583

B. Composite (derived) metric values:

- Number of training tasks remaining: $\theta_1(k) = N - k$
- Quantile bound for the maximum time for one task: $\theta_2(k) = Q_{1\frac{a}{2}}(\boldsymbol{\theta}_5)$
- Cores needed to do the trainings in parallel:

$$\theta_4(k) = \left\lceil \frac{\theta_1(k) \cdot \theta_2(k)}{\frac{T_{max}}{\varepsilon} - \theta_3(k)} \right\rceil$$

2. Then, it is possible to calculate the composite metric values, such as number of training task remaining, or the quantile bound for the maximum time for one task. Also, it is possible to estimate the number of cores needed to do the trainings in parallel.
3. The next step is to provide some fixed parameters from the application owner or the adaptation engineer. For this example, it would be the deadline, which is the wall clock time to finish all trainings, and the possible tolerable deviation from that deadline. Also, the price of the cheapest usable VM will be useful to be included in the utility function. These parameters are:
   A. Wall clock time to finish all training: $T_{max} = 3600s$

   B. Tolerance for deviation: $\varepsilon = 0.85$

   C. Price of the cheapest usable VM: $P^-$
4. The last composite metric value, so the number of cores needed to do the trainings by deadline, is then used in the metric constraint. This constraint can trigger the reconfiguration if the current number of cores is not enough to finish on time. What is more, it is also used in the constrained problem, to make sure that the reconfigured deployment will have enough cores.

   In detail, according to the constraint, it does not wait too long even if the cost higher. The number of cores should not be less than the tolerance $\varepsilon$ on the estimate:

$$c_1 \cdot c_2 \geq \varepsilon \cdot \theta_4(k)$$

   Where:

   A. $c_1$: The number of instances.
   B. $c_2$: The number of cores.
   C. $\theta_4(k)$: The estimated number of cores.

5. Finally, the utility function can be constructed. It should consider two dimensions. The first dimension is related to the cost, and it represents the desire to keep the cost as low as possible. The second dimension is the performance measured as finishing time. One can use a sigmoid function to express the performance aspect. The 0.5 utility value can be then assigned to the configuration that ensures finishing on time, and the utility should be higher for configurations that allow finishing earlier, and lower for configurations that will lead to finishing after the deadline. The sigmoid performance utility function can look like the following:

$$\mathrm{u}_2(\boldsymbol{c} \mid \boldsymbol{\theta}(k), \boldsymbol{\varphi}) = \frac{1}{1 + exp(a \cdot \frac{\left\lceil \frac{\theta_1(k) \cdot \theta_2(k)}{c_1 \cdot c_2} + \theta_3(k) - T_{max} \right\rceil}{T_{max}})}$$

   Where:

   A. $c$: The application configuration.

B. $\theta$: The measurements.
C. $\varphi$: The parameters.
D. $\theta_1$: The number of trainings left to do.
E. $\theta_2$: The percentile bound of executing one task.
F. $\theta_3$: The elapsed time.
G. $c_1$: The number of instances.
H. $c_2$: The number of cores.
I. $T_{max}$: The deadline.

The cost dimension can be modelled by simple fraction of on-demand price of the less expensive VM and the on-demand price of the proposed configuration. This will ensure that the function value stays within zero and one interval and reflects the fact that the cheapest possible deployment should give the highest cost utility value. The formula for this function is presented:

$$\mathrm{u}_2(\boldsymbol{c} \mid \boldsymbol{\theta}(k), \boldsymbol{\varphi}) = \frac{P^-}{c_1 \cdot Price(M)}$$

Where:

A. $c$: The application configuration.
B. $\theta$: The measurements.
C. $\varphi$: The parameters.
D. $P^-$: Price of the least expensive VM.
E. $c_1$: The number of instances.
F. $M$: The selected VM worker type.

The overall utility function formula can be the affine combination of two dimensions, where weights of two dimensions sum to one. It is the application owner's decision to put the accurate weights so to indicate which aspect is more important in overall utility of the application deployment.

$$U(\boldsymbol{c} \mid \boldsymbol{\theta}(k), \boldsymbol{\varphi}) = w_1 U_1(\boldsymbol{c} \mid \boldsymbol{\theta}(k), \boldsymbol{\varphi}) + w_2 U_2(\boldsymbol{c} \mid \boldsymbol{\theta}(k), \boldsymbol{\varphi})$$

### 3.1.3 MORPHEMIC Monitoring

#### *3.1.3.1 Monitoring specification input to EMS*

The monitoring of applications running on top of MORPHEMIC is based on the use of the Event Management System (EMS). The EMS receives as its input the CAMEL file which includes, among other deployment details, the metrics which should be monitored. The functionality provided by the EMS is delivered through the EMS Server and the EMS client(s). The EMS Server collects raw monitoring metric values and performs groupings, as well as window and time-based processing on these, also generating composite metrics where this is necessary.

For the EMS to identify the raw metrics to be used, as well as the composite metrics which are generated, the CAMEL deployment file is consulted. When the deployment file is parsed through an appropriate translator mechanism (integrated into the EMS), the EMS creates monitoring probes (one or more EMS clients) on the processing infrastructure (VMs, BYONs, containers) to be monitored.

Subsequently, monitoring probes send monitoring metrics to the EMS according to the specifications set in the CAMEL file. The EMS is responsible for providing the final monitoring metric values to the Metasolver and any other Upperware component.

#### *3.1.3.2 Novel EMS capabilities*

In addition to the monitoring capabilities which have been traditionally offered by the EMS – and which have been enhanced as part of MORPHEMIC – the EMS is further able to differentiate its behaviour depending on the nature of the processing infrastructure underneath. Specifically, when resource-limited (RL) processing nodes are detected (the definition of which is configurable) the EMS will not install an EMS client on the device nor a JRE8[17], but will rather rely on the raw monitoring metric input provided by Netdata[18]. This action is taken to further reduce the processing strain on hosts with small processing power (e.g edge devices) which need to fully utilize their processing power to perform any tasks allocated to them.

Moreover, the EMS is now able to function in either two-level or three-level monitoring topologies. Two-level monitoring topologies are comprised of an EMS server which manages any number of monitoring nodes. Three-level monitoring topologies are comprised of the EMS server, Aggregators and any number of monitoring nodes. Aggregators are determined from the monitoring nodes (excluding RL-nodes) which belong to a particular availability zone or cloud provider (or belong to the 'default' cluster). In the case of a three-level topology, it is the Aggregators which send monitoring metrics to the EMS server.

The MORPHEMIC EMS also takes action to maintain the availability of monitoring nodes, attempting their recovery when a connection to the EMS client (or a connection to Netdata - the monitoring metric provider installed even in RL nodes) is lost. Recovery will also be attempted in three-level topologies, when a non-RL node leaves the cluster and does not join within a configurable interval. To recover the EMS client / Netdata agent, the EMS will open an SSH connection, try to kill the malfunctioning component (if still running) and try to restart it. More details on the operation of the EMS can be found in the README file of the component[19].

#### 3.1.4 MORPHEMIC Adaptation Example

To illustrate the MORPHEMIC adaptation, we can use as an example the deployment of our two test applications created for MELODIC project: GENOM and FCR [20] [21]. First, we need a CAMEL model containing aspects of deployment, requirement, metric and scalability specified by the user. After converting it in CAMEL_EDITOR using Meta Data Schema with annotations, we can upload the XMI to our platform and MORPHEMIC will start the validation process. For GENOM we will be asked to provide sensitive variables. After the model is successfully uploaded, we can move to the next stage and select cloud definitions or add BYON (Bring Your Own Node- private node provided by user), EDGE[22] definitions if we are using them for deployment. Now we are ready to start our application. Deployment on MORPHEMIC platform contains several successive stages: Fetching offers, Generating Constraint Problem, Reasoning, Deploying and ends up with Updating application state. Every stage must be successful in order to proceed to the next. When application is deployed, we can move to the test phase.

To test FCR application we need to use jmeter tool[23] . In MORPHEMIC GUI go to "My Application" tab on the left and find the IP of LOAD_BALANCER component. On the machine where you've got jmeter installed go to:

```
cd jmeter/apache-jmeter-5.5/bin/
```

and run:

```
./jmeter -n -t fcr.jmx -Jhost={LOAD_BALANCER_IP} -Jusers=100
```

We can use Grafana, a visualization tool, at http://{PUBLIC_MELODIC_IP}:3000 and already prepared dashboards for FCR or GENOM to see the metrics and predictions. It allows us to visualize and monitor all our data in one place using graphs.

## 3.2 DevOps Materials

The MORPHEMIC platform offers a comprehensive suite of tools and techniques to support DevOps practices. In this section, we will explore the foundational MORPHEMIC concepts of DevOps, starting with "Getting started with the

[17] JDK 8 and JRE 8 Installation
[18] Netdata's Official GitHub Repository
[19] Testing of new EMS features documentation
[20] The test cases directory of the testing repository.
[21] The Genom application demo presentation for the MELODIC project.
[22] MORPHEMIC: A complete paradigm from cloud to edge
[23] The official Apache Jmeter website.

CAMEL 3.0 modelling language" which will introduce you to the CAMEL 3.0 modelling language used for specifying and designing complex systems. We will then move on to "Getting started with MORPHEMIC," where you will learn how to use the MORPHEMIC platform to automate and optimize your DevOps processes. Finally, we will cover "Polymorphic modelling in MORPHEMIC," which is a powerful technique for modelling complex systems with multiple variations. By the end of this section, you will have a solid understanding of the DevOps methodology and the various tools and techniques used to implement it.

### 3.2.1 CAMEL "Getting Started"

The project website is currently being updated with the relevant documentation guides to Getting Started with CAMEL. There we explain what CAMEL is, we supply information about its two main editors and how they can be installed and then we shortly analyse CAMEL explaining what should be involved in a CAMEL model of a polymorphic application and which links to follow in order to find more information about the modelling of the respective needed CAMEL aspects/domains in that model. The landing page also clarifies which CAMEL features to utilise during application modelling as not all CAMEL features are or will be supported by the MORPHEMIC platform. This landing page ends by forwarding the reader to a confluence page explaining how a particular MORPHEMIC use-case has been modelled with CAMEL and supplying a link to all its information sources (see Section 3.1.1).

### 3.2.2 MORPHEMIC "Getting Started"

The process of getting started with the MORPHEMIC platform involves installing and configuring it, as detailed in MORPHEMIC deliverables *D4.6 Test report for prototype release* and *D5.2 User Interface Guidelines*. Here are the steps to follow:

1. Provision and access the machine for MORPHEMIC installation, ensuring that the required ports are open to allow inbound traffic to our services. This can be done by establishing an SSH connection to the virtual machine (VM) and accessing the user's home directory.
2. Download the installation file by cloning the Git repository and checking out the desired release branch.
3. Run the installation script to install MORPHEMIC.
4. After installation, perform some basic configuration to make the MORPHEMIC platform available for usage. Load the system profile that includes aliases specifically created for MORPHEMIC.
5. To start the Docker containers and establish integration between them, use one of the aliases from the loaded profile. You can simply run the command "drestart" to achieve this.
6. Use the alias "mping" to check if the platform started correctly. This will verify the status of all Docker containers. If they are all in the "OK" status, you can proceed to the next step, which involves configuring the LDAP policy and creating a new LDAP user.

The MORPHEMIC project website is currently being updated with documentation to help users and businesses get started with the platform as described above.

### 3.2.3 MORPHEMIC Polymorphic Modelling

The MORPHEMIC platform facilitates polymorphic adaptation, allowing for dynamic modification of application variants or execution units based on performance analysis. As an example, MORPHEMIC can decide to deploy the Docker version of an application component while it was previously running without virtualization. This functionality relies on information present in the application model (CAMEL model), which enables polymorphic optimization.

To incorporate polymorphic modelling, metadata needs to be inserted into a specific section of the CAMEL model, as explained in the relevant materials in Section 3.1.1. The application model should include information that allows the MORPHEMIC platform to consider different variants supported by the application component or execution units (such as GPU or FPGA) where the component can run.

For further technical documentation, the project's internal Confluence documentation repository[24] provides in-depth insights into the architecture of polymorphic adaptation.

[24] MORPHEMIC Polymorphic Adaptation Process (requires free registration)

# 4 Conclusions

This document has described the training materials which are under use by members and contributors of the MORPHEMIC project’s Work Package 7, Outreach, and has provided a comprehensive categorization of these materials into their target trainee audiences. The MORPHEMIC deliverable *D7.3 Training plan and activities*, will provide complete, in-depth information about the training plan’s design approach, as well as all the training activities and additional training opportunities presented during the course of the project. The results of the training programme’s activities will be included in the deliverable *D7.2 Final Dissemination and Communication Report and Plan*.

# 5 Appendix A – CAMEL Materials

## 5A CAMEL 3.0 Information Sources

1. Excel file that covers a detailed analysis of CAMEL 3.0 classes. It includes both a description of a CAMEL class but also the syntax for specifying its elements in the CAMEL textual format. The latest version of this file can be found here: https://gitlab.ow2.org/melodic/camel/-/blob/camel-3.0.4/camel/camel.dsl.ui/input/documentation.xlsx.
2. Folder (https://gitlab.ow2.org/melodic/camel/-/tree/camel-3.0.4/camel/examples) in CAMEL's Gitlab containing:
   a. example CAMEL application models mapping to use cases from previous related projects
   b. metric, type & unit template models
   c. MELODIC's metadata model
3. Confluence Pages related to CAMEL Modelling. As CAMEL 3.0 builds incrementally on CAMEL 2.5, the modelling pages include the modelling with respect to CAMEL 2.5 domains/aspects as well as extensions made to these domains by CAMEL 3.0.
   a. CAMEL 2.5 Modelling: https://confluence.7bulls.eu/display/MEL/11+Modelling containing following subjects / sub-pages:
      - Deployment Modelling
      - Requirements Modelling
      - Resource requirements
      - Metric Modelling
      - Utility function modelling - general guidelines
      - Modelling of Utility functions and constraints
      - Proposition of the utility function for the Genom and FCR use cases
      - Data Modelling
      - Unit Modelling
      - Value Type Modelling
      - Location Modelling
      - Scalability Rule Modelling
      - CAMEL 2.5 features supported by Upperware in release 2
      - CAMEL FAQ
   b. CAMEL 3.0 Extensions:
      - CAMEL 2.5 vs. 3.0
      - CAMEL Configuration Alternatives
      - Attribute Modelling Extensions
      - Deployment Modelling Extensions
      - Requirements Modelling Extension
      - Metadata Modelling Extension
      - Metric Modelling Extensions
      - MORPHEMIC Support to CAMEL 3.0 Features
4. Editor-related Information:
   a. Textual CAMEL 3.0 Editor Installation: CAMEL 3.0 Eclipse Editor Installation & Execution
   b. CAMEL graphical editor, CAMEL Designer, user guide: Camel Designer User guide
   c. CAMEL graphical editor, CAMEL Designer, installation instructions: https://github.com/Modelio-R-D/CamelDesigner/wiki/Installation
   d. CAMEL graphical editor, CAMEL Designer, usage instructions: https://github.com/Modelio-R-D/CamelDesigner/wiki
5. CAMEL Meta-Model & Source Code: https://gitlab.ow2.org/melodic/camel/-/tree/camel-3.0.4/camel/camel
6. Overall CAMEL source code including textual editor: https://gitlab.ow2.org/melodic/camel/-/tree/camel-3.0.4/camel

## 5B CAMEL 2.5 vs 3.0

CAMEL 3.0 is a backwards-compatible extension of CAMEL 2.5, i.e., of the main CAMEL version that was adopted by the MELODIC platform. In this manner, any CAMEL 2.5 (application) model is still a valid CAMEL 3.0 model. This enables migrating easily any kind of application model in CAMEL 2.5 towards CAMEL 3.0.

The following table summarises the extensions that have been made to CAMEL 2.5 in order to produce CAMEL 3.0. The first column covers the domain extended in CAMEL; the second column covers the actual extension made to that domain while the third column explains the rationale of this extension.

*Table 1 CAMEL 2.5 feature extension summary*

| Domain | Extension | Rationale |
|---|---|---|
| Core | Enhancement of attributes to include their minimum and maximum values | Can specify constraints on the minimum or maximum value of attributes apart from their exact value. This is quite useful for matching resource requirements (within a CAMEL model as range constraints on attributes like number of cores) with resource capabilities (drawn from a provider site or other kind of sources). |
| Deployment | Incorporation of a reference to a respective requirement for a Communication | Ability to refer to constraints on the quality of communication (e.g., latency) specified via a communication requirement. |
| Deployment | Incorporation of a reference from a configuration to its set of requirements | A component configuration should have its own requirement set as different configuration kinds tend to differ with respect to their requirements. This does not preclude the existence of common requirements across all configurations. In that case, such requirements are directly referenced by the respective application component. This is an essential feature to support polymorphic application modelling. |
| Deployment | Introduction of new configuration kinds | As MORPHEMIC aims at supporting new kinds of resources for hosting application components, such resource kinds (e.g., hardware-accelerated ones) should correspond to new configuration kinds for application components (e.g., image configurations). This facilitates polymorphic application modelling as it introduces new capabilities for expressing new kinds of configurations for application components. |
| Deployment | Introduction of control-flow relationships between application components | These relationships enable making better decisions in terms of application placement as they can indicate the order of execution of application components, such that we can save resources and create/allocate them only when needed. Such relationships are, of course, relevant mainly for workflow-based applications. |

| Domain | Extension | Rationale |
|---|---|---|
| Requirement | Introduction of communication requirements | Need to introduce a new requirement kind which can include constraints on the quality of communication (e.g., latency, jitter) between two or more application components. |
| Metric | Object context reference to communication | The object that is being measured in an object context within a metric context can be a component, data or now the communication between components, giving the ability to define metrics over the communication quality (like average or maximum latency). |
| Metric | Metric variables can now refer to and encapsulate metrics | Metric variables can now refer to both formulas and metrics. In the first case, they can be computed by evaluating these formulas at the beginning of the application deployment. In the second case, they can be computed through metric prediction over the current solution alternative examined by a MORPHEMIC Solver once enough data for the prediction are available. This enables increasing deployment reasoning accuracy as it can evaluate solution alternatives based on the predicted impact that they can have on application quality. |
| Metric | Introduction of window processing (capabilities) | Window processing represents a new ability to be featured by MORPHEMIC in terms of performing certain kinds of window pre-processing, i.e., processing measurements (e.g., filter or group them) before they can enter a window. Only the remaining measurements in the window after this pre-processing will be used for computing the respective composite metric's value. This new feature makes metric specification richer covering more metric modelling cases while it also assists to reduce the modelling effort for specifying metric hierarchies for applications. |
| Metadata | Introduction of the *implemented* attribute in metadata objects | The MELODIC Metadata Schema (MDS) is quite extensive to cover any kind of conceptualization relevant to Cloud services and big data. The MORPHEMIC platform supports only a part of MDS which is signalled to the platform user via this attribute. |

## 5C MORPHEMIC Support to CAMEL 3.0 Features

CAMEL is a very rich language covering multiple domains that are relevant to the application management lifecycle. In fact, CAMEL has been designed in order to be platform-agnostic and thus not bound to any platform. As such, it can be potentially exploited by any prototype or commercial platform. Due to this fact, not all features of CAMEL are supported by the MORPHEMIC platform.

By breaking down these features into CAMEL 2.5 features and extension-based features brought by CAMEL 3.0, the reader can find the level of support for CAMEL 2.5 features in CAMEL 2.5 features supported by Upperware in release 2. In the following table, we supply the level of support for the new features introduced by CAMEL 3.0:

*Table 2 CAMEL 3.0 Feature support status in MORPHEMIC*

| Domain | Extension | Support |
|---|---|---|
| Core | Range-based constraints in attributes | Will be supported in the final MORPHEMIC platform release. |
| Deployment | Reference to communication requirements in communications | It will not be supported. |
| Deployment | Mapping of application configurations to their own requirements | Will be supported in the final MORPHEMIC platform release. |
| Deployment | New configuration kinds: ContainerConfiguration & ImageConfiguration | It will not be supported directly. Container and image-based configurations are covered through other configuration kinds. |
| Deployment | Introduction of control flow relationships between application components | Will be supported in the final MORPHEMIC platform release. |
| Requirement | Introduction of communication requirements | It will not be supported. |
| Metric | Extension of *ObjectContext* to refer to a communication | It will not be supported. Communication-related metrics will not be supported by the platform. |
| Metric | Association of metric variables to metrics (for exploiting metric predictions in deployment reasoning) | Will be supported in the final MORPHEMIC platform release. |
| Metric | Window processings | Will be supported in the final MORPHEMIC platform release. |
| Metric | Two new kinds of window size types | It will not be supported. |
| Metadata | Insertion of *implemented* attribute in metadata objects | It is supported by the MDS editor and CAMEL. It is mainly used for informing application modellers which MDS objects (e.g., concepts and properties) can be utilised in CAMEL models as they are implemented by the MORPHEMIC platform. |

Based on the above analysis, an application modeller is advised to utilise only those features that are or will be supported in the MORPHEMIC platform in his/her CAMEL models.

### a. Attribute Modelling Extensions

Attributes in CAMEL enable to specify characteristics of features like CPUs, GPUs, and main memory at the resource level or environment at the platform level. These attributes are exploited in order to specify constraints on the values of these feature characteristics in resource or platform requirements in CAMEL. In this respect, an attribute is a combination of a feature characteristic (e.g., number of cores) and its constraint.

In CAMEL 2.5, constraints were quite simple. They indicated the actual value that the feature characteristic needs to take. Thus, they were mapping to equality constraints. Although attributes were associated with a value type, such a value type indicates the allowed values that the attribute can take and the allowed (required) values in the equality constraints. This modelling led to the need to simulate a single feature characteristic to multiple attributes when range constraints had to be specified for this characteristic. For instance, if the feature characteristic was the number of cores, then we had to create two attributes, minNumberOfCores and maxNumberOfCores. So, in case we had to specify that the number of cores should be between 2 and 4, then we had to specify that the value of minNumberOfCores attribute equals 2 and the value of maxNumberOfCores attribute equals 4. Further, as each attribute has to be semantically annotated via the MetaData Schema (MDS), there was also a need to specify two respective properties in MDS. As such, this issue was leading to a very extensive version of MDS as well as to verbose CAMEL models (requiring specifying two attributes per each range-based feature characteristic constraint).

The modelling of the above case in CAMEL 2.5 could be expressed as follows:

```
resource requirement SmartDesignResourceReqs{

    feature cpu{

        [
MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU ]

        attribute mincores


 [MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU.h
 asMinNumberofCores] :

          int 2 UnitTemplateCamelModel.UnitTemplateModel.Cores
        attribute maxcores

 [MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU.h
 asMaxNumberofCores] :
          int 4 UnitTemplateCamelModel.UnitTemplateModel.Cores

    }

}
```

Through CAMEL 3.0 and its attribute extension, attributes can now have minimum and maximum values. In this respect, it is possible not only to specify equality constraints on feature characteristics but also range-based constraints where a range-based constraint can cover both upper and lower bounds. And this can be done just with a specification of a single attribute. Further, it is possible to indicate whether the bounds are inclusive or exclusive in the range-based constraint.

Continuing our previous example, the previous resource requirement could be specified in CAMEL 3.0 as follows:

```
resource requirement SmartDesignResourceReqs{

    feature cpu{

        [
MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU ]
```

```
        attribute cores

 [MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU.h
asNumberofCores] :
          [int 2, int 4] UnitTemplateCamelModel.UnitTemplateModel.Cores
    }
}
```

As can be seen above, our original requirement can now be expressed via a single attribute which maps to a single property in MDS. As such, MDS size becomes smaller. Further, the respective requirement is more compact, thus making the CAMEL model more laconic.

Please also check the difference from the previous example in terms of an attribute definition. After specifying the semantic annotation through MDS in the definition of an attribute and before supplying the respective unit (of measurement), we can see that instead of supplying a single value - int 2, we specify a range of values - [int 2, int 4]. This actually signifies the difference between equality constraints and range-based constraints.

Let's cover here all possible cases for the constraint part in an attribute definition:

- <type> <val>: here we specify a single equality constraint specifying that the attribute should have a specific <val> value of a specific <type>. For instance, if we desire to specify that the attribute should have an integer value of 3, this could be done as follows: "int 3". If we desire a double value of 3.5, then we will specify: "double 3.5".
- [<type> <val>: here we specify only the lower bound of a range which is inclusive. For instance, "[int 4" indicates that the value of the attribute should be greater or equal to 4 (i.e., attribute >= 4).
- (<type> <val>: same as the previous case but the lower bound is not inclusive. For instance, "(double 2.5" means that the value of the attribute should be greater or equal to 2.5 (i.e., attribute >= 2.5).
- , <type <val>]: here we specify only the upper bound of a range which is inclusive. For instance, ", int 4]" means that the value of the attribute should be less or equal to 4 (i.e., attribute <= 4). Please be aware of the ',' character. It needs to be given so as to highlight that we move to express the upper bound of the constraint.
- , <type> <val>): same as the previous case but the upper bound of the range is not inclusive. For instance, ", double 4.5)" means that the value of the attribute should be less to 4.5 (i.e., attribute <= 4.5).
- [<type1> <val1>, <type2> <val2>]: here we specify a complete range constraint with both bounds inclusive. For instance, [int 2, int 4] means that attribute value should be greater or equal to 2 and less or equal to 4 (i.e., 2 <= attribute <= 4). You can now image that we can have combinations that might have one of the two ranges not inclusive. Some examples are given below:
    - (int 2, int 5]: 2<attribute<=5
    - (int 2, int 5): 2<attribute<5
    - [int 2, int 5): 2<=attribute<5
    - [int 2, int 5]: 2<=attribute<=5

We finish by providing an example with multiple attribute constraints for a resource requirement. This example indicates that the number of CPU cores should be from 2 to 8 inclusive while the main memory size should be at least 8192 Megabytes.

```
resource requirement SmartDesignResourceReqs{
    feature cpu{
        [
MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU ]
        attribute cores
[MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.CPU.h
asNumberofCores] :
          [int 2, int 8] UnitTemplateCamelModel.UnitTemplateModel.Cores
        attribute ram

[MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Processing.Memor
y.ProcessingMemory.RAM.TotalMemory] :
          [int 8192 UnitTemplateCamelModel.UnitTemplateModel.MegaBytes
    }
}
```

## b. Deployment Modelling Extensions

### I. Communication Requirement Reference

As application components can communicate, the frequency and size of the data exchanged in such communication can have an impact on application performance cost. In this respect, we can place requirements on a communication in order to guarantee that either the components can be close to each other or have a communication link with the appropriate quality. Such requirements refer to the quality of the communication (e.g., latency, jitter, etc.). Once they are in place, they can influence the placement of the respective application components such that the respective communication link between them respects such requirements. In this respect, the performance of the application could be guaranteed.

We will refer to the way communication requirements can be expressed in Requirements Modelling Extension. Here we just indicate that in CAMEL 3.0 we can specify communication requirements as well as attach them to a specific communication. The latter has been enabled by extending the deployment meta-model in CAMEL such that there can be a reference from a *Communication* to its *CommunicationRequirement*.

For instance, suppose that a specific *CommunicationRequirement* in a CAMEL model has the name *CommReq1* and has been defined in a *RequirementModel* named *ReqModel*. Then, we can refer to and thus attach this requirement in a specific *Communication* as follows:

```
communication StorageManagerToInformationService from StorageManager.InfSerPortReq
to InformationService.InfSerPort req ReqModel.CommReq1
```

Thus, we use the word "req" to indicate a reference to a communication requirement and then we refer to this requirement through the known pattern of "<nameOfContainingModel>.<nameOfElement>" (e.g., ReqModel.CommReq1).

### II. Requirements for Configurations

Single-form applications contain components with a single configuration. All these components can have various sorts of requirements, some of which could be specific to their configuration. But, as each component has only one configuration, it is still suitable to map this component to all these requirements.

In the case of polymorphic applications, the situation now changes. Each component of the application can have multiple configurations. Although some of the requirements could hold for any kind of configuration and thus be attributed to the component itself, there can be specific requirements which should now be attributed to specific configurations. For instance, an FPGA-based configuration could have different resource requirements (e.g., imposed on FPGA features like cores) with respect to a container-based configuration (e.g., number of cores for CPU) for the same component.

CAMEL 2.5 allows the mapping of multiple sorts of requirements only to application components. Thus, it is suited only for single-form applications. As such, CAMEL 3.0 has extended CAMEL 2.5 so as to enable mapping requirements to both components and their configuration, thus allowing for the specification of polymorphic applications.

Syntactically, we can showcase this extension by considering the example where we have an application component that needs to be deployed in a public cloud (*PublicRequirement* requirement) and can be scaled from 1 to 10 instances (*HorizontalScaleWorker* requirement). This component has two configurations:

- An FPGA-based with two requirements: one resource requirement named *WorkerReqs_FPGA* (1 FPGA unit and 8 CPU cores) and one image requirement named *Ubuntu18_FPGA* (for a particular OS image)
- A container-based with two requirements: one resource requirement named *WorkerReqs* (2-4 CPU cores and 8100-10072MBs of main memory) and one image requirement named *Ubuntu_18* (for a particular OS image

All these requirements are specified in the requirement model *ReqModel* while the components are specified in a deployment model named as *DepModel*. The respective content for the application deployment model in CAMEL 3.0 could be as follows:

```
deployment model DepModel{
...
  software Worker{
    requirements WorkerRequirementSet
    required host WM_WorkerHostReq
    script configuration WorkerScript{
      download 'x' install 'y' configure 'z' start 'w'
      requirements Worker_Script_Reqs {
        resource ReqModel.WorkerReqs
        image ReqModel.Ubuntu18
      }
      attribute category_t1_docker
[MetaDataModel.MELODICMetadataSchema.Context_Aware_Security_Model.SecurityContextEle
ment.Object.SoftwareArtefact.softwareCategory] : string "DOCKER"
      attribute category_t1_hardware
[MetaDataModel.MELODICMetadataSchema.Context_Aware_Security_Model.SecurityContextEle
ment.Object.SoftwareArtefact.softwareCategory] : string "CPU"
    }
    image configuration WorkerImgConfig{
      id 'myFPGAImage1'
      requirements FPGA_Reqs{
        resource ReqModel.WorkerReqs_FPGA
        image ReqModel.Ubuntu18_FPGA
      }
      attribute category_t2_hardware
[MetaDataModel.MELODICMetadataSchema.Context_Aware_Security_Model.SecurityContextEle
ment.Object.SoftwareArtefact.softwareCategory] : string "FPGA"
    }
    required communication WorkerPort port 64738 mandatory
  }

  requirements WorkerRequirementSet{
    horizontal scale ReqModel.HorizontalScaleWorker
    provider ReqModel.PublicRequirement

   }
 }
```

Some things to be noted from the above example are the following:

- the requirements for a component hold for all its configurations but potentially could be overridden by specific requirements in the configurations. Further, the *RequirementSet* element (named as *WorkerRequirementSet* in our case) that gathers these component requirements is not defined inline inside the component's definition but outside (within the deployment model). So the component element (*Worker* in our case) just references this *RequirementSet* (see *requirements WorkerRequirementSet* inside the definition of the *Worker* element). This is done based on the rationale that a specific requirement set can be shared between multiple application components.
- the requirements for a configuration are also gathered in the form of a *RequirementSet* element. However, this time, this element is defined within the definition of the configuration element. For instance, inside the definition of *WorkerScript* (a *ScriptConfiguration* element) we can see the reference (see *requirements* reference) to a *RequirementSet* element (named as *Worker_Script_Reqs*) which is defined inline. This is a modelling choice that relies on the fact that a set of requirements that holds for a specific configuration will be different from that of another configuration (for the same component) so there is no need to reuse it.

## III. New Configuration Kinds

CAMEL 3.0 extends CAMEL 2.5 with the introduction of two new Configuration kinds, *ContainerConfiguration* and *ImageConfiguration*. The first can be used for specifying container-based configurations for application components while the second for specifying FPGA-based configurations for application components. All the configuration kinds in CAMEL 3.0 are analysed, compared and exemplified in CAMEL Configuration Alternatives. As the set of configuration kinds has been extended, this enables covering additional cases in component configuration modelling, further assisting in the definition of components with multiple configurations, thus facilitating the specification of polymorphic applications.

## IV. Control Flow Relations between Components

Apart from location, communication and hosting relationships between components, which are covered in CAMEL 2.5, there is also a need to cover control-flow relationships, especially in the context of workflow-based applications. Such control-flow relationships can determine the (partial or complete) execution order of application components. The knowledge of this execution order can be crucial for the operational cost of a cloud application and its deployment reasoning.

For instance, if we know that one application component B is executed after another one A, we do not need to create a VM for hosting this component (B) until it is the time to execute it. Even more, we could decide to collocate A and B in the same VM such that we can save some provisioning time. Indeed, we could create one VM that covers the maximum from the resource requirements for both components and we could install both components there. Then as the components are executed in completely different time spaces, there will be no case of resource interference between them. As such, we do not need to create two different VMs (in case we do not know this relationship) or one VM and then another one (case where we deploy the components based on their execution order), saving precious provisioning time along with operational cost.

Based on the above analysis, it could be argued that the MORPHEMIC platform could utilise the specification of the application workflow in order to extract the execution order (control-flow) of the application components without extending CAMEL. However, this creates various issues: (a) this would create a dependency on the MORPHEMIC platform with respect to the adoption of a specific workflow language, (b) it might also require supporting multiple workflow languages and not just one to cover multiple cases, (c) it would change significantly the way the platform works but also the kind of input that it requires from the user, (d) potentially only some part of the workflow could be of interest (with a focus on specific control flow constructs) and not the whole workflow specification. As such, it was decided to extend CAMEL 2.5 so as to clearly specify the control-flow relationships between application components in a workflow-language-agnostic way also catering for the specification and usage of just one model originating from the user, the actual CAMEL model of his/her application. Thus, the MORPHEMIC platform will rely on a self-contained model of the user application that contains all kinds of relationships between application components such that it can facilitate the deployment reasoning so as to infer the best possible deployment solution for this application.

Motivated by the existing wisdom in the workflow management domain/area, CAMEL was extended so as to cover basic control-flow relationships between application components which map to well-known simple workflow constructs (for control-flow). These constructs have been shown that could be utilised to simulate any kind of more sophisticated or composite workflow construct and thus for the modelling of any kind of workflow. In the following, we analyse these control-flow relationships by also supplying respective examples of their usage in CAMEL models.

*PRECEDES*. This control-flow relationship relates one application component to multiple ones. Its semantics is that the first application component precedes the others in the execution order. However, it does not clarify what is the execution order / control-flow relationships between these latter/other components. Suppose that we have application component A that precedes in execution order with respect to application components B, C and D. The PRECEDES relationship could be expressed in the deployment model of the application as follows:

```
precedes A_BCD{
  component A
  preceded components [B, C, D]
}
```

Please note that we need first to refer to the component A (that precedes in execution order) and then to the preceded components that are mentioned in a set-like manner. The respective relation is classified as PRECEDES (see *precedes* key-word) and has the name of *A_BCD*.

SEQUENCE. This control-flow relationship expresses a sequential execution order for a set of components. This means that the referenced components in this relationship are executed one after the other. The actual order of reference for these components correlates to their sequential execution order. Suppose that we have components A, B, C, D where component A is executed before B, B before C and C before D. All these pair-wise relationships between these 4 components could be expressed through a single SEQUENCE relationship as follows:

```
sequence ABCD [ A, B, C, D]
```

So, we specify a single sequence element (of type SEQUENCE) with the name ABCD through which we refer to all the components (A, B, C, D), which are executed sequentially, in a set-like manner.

PARALLEL. This control-flow relationship expresses a parallel execution order for a set of components. This means that the referenced components in this relationship are executed in parallel. Suppose that we have components A, B and C which are executed in parallel. We can express a single PARALLEL relationship between them as follows:

```
parallel ABC [A, B, C]
```

CONDITIONAL. This control-flow relationship expresses a conditional execution order between two components. The semantics is that one component from the two will be picked up for execution depending on the result of assessment of a specific condition. So, if the condition is true, then the first component will be executed. Otherwise, if it is false, the second component will be executed. It is considered that the condition is an optional part of the relationship in terms of the respective knowledge that should be conveyed through this relationship that could be exploited by the MORPHEMIC platform. This is due to the fact that the condition could relate to a specific variable of the application workflow. As such, as this variable will not be manipulated by the platform, the respective condition including it is not needed. However, in case the condition relates to a specific metric that is somehow measured by the platform, then this condition could be explicitly modelled. In any case, we argue that such a condition is not actually needed based on the rationale that its respective contribution to the deployment reasoning is that it will map to allocating the two related components in the same VM as only one of them will be executed.

Suppose that we have components A and B that are conditionally executed. This relationship can be modelled via CAMEL as follows:

```
conditional AB{
  first A
  second B
}
```

As can be seen, we specify a CONDITIONAL element named *AB* and we refer to the two related components based on *first* and *second* clauses.

SWITCH. This is the last kind of control-flow relationship. Its semantics is that only one of the referenced components will be executed depending on the value of a specific element. In case this element is a variable, as indicated in the above relationship kind, it will not be modelled. In case this element is a metric that can be computed by the MORPHEMIC platform, then it could be modelled by referring to its *MetricContext*. In the latter case, we can also model the values that this metric can take as Strings. The order of these values would need to be the same as the order of the application components that are enabled by these values. For instance, if component A is enabled by the value "A" and component B by the value "B", then if components A and B are mentioned based on this order in the switch relationship, this means that the values should be mentioned in the order of "A" and "B" in that case.

Suppose that we have components A, B and C that are participating in such a relationship (e.g., A will be executed if workflow variable X has the value of 0, B if the value is 1 and C if the value is 2). Then, the respective CAMEL model part that will model this relationship could be the following:

```
switch S_ABC{
  components [A, B, C]
}
```

COMPLETE EXAMPLE: Suppose that the workflow of the application is the following:

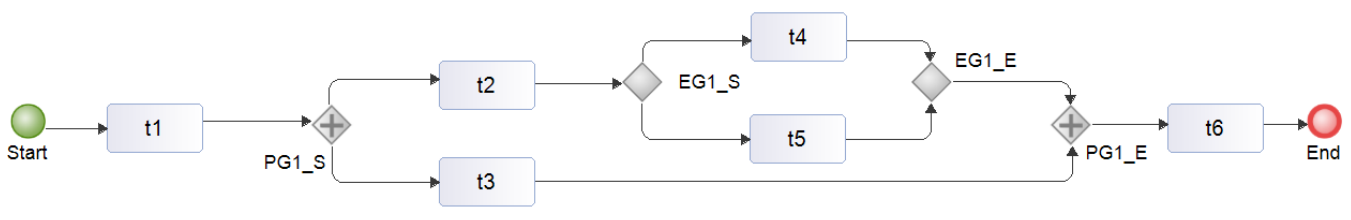

*Figure 2 Sample application workflow*

In this workflow, component t1 is executed first. Then we have two sub-flows that are executed in parallel. The second sub-flow corresponds to the execution of component t3. The first sub-flow maps to the execution of component t2 which is followed by a conditional execution of components t4 and t5. Both sub-flows are followed by the execution of the last application component, t6.

The respective part of the deployment model of the application (expressing all the necessary control-flow relationships) will be the following (as expressed in CAMEL 3.0):

```
deployment model DepFlowModel{
  software t1{
    cluster configuration cc1{
    }
  }

  software t2{
    cluster configuration cc2{
    }
  }

  software t3{
    cluster configuration cc3{
  }
  }

  software t4{
    cluster configuration cc4{
    }
  }

  software t5{
    cluster configuration cc5{
  }
  }

  software t6{
    cluster configuration cc6{
    }
  }

  precedes t1_t2{
    component t1
    preceded components [t2]
  }

  precedes t2_t4{
    component t2
    preceded components [t4]
  }

  parallel t2_t3 [t2, t3]

  conditional t4_t5{
    first t4
    second t5
  }

  precedes t3_t6{
    component t3
    preceded components [t6]
  }

  precedes t4_t6{
    component t4
    preceded components [t6]
  }

  precedes t5_t6{
    component t5
    preceded components [t6]
  }
}
```

## c. Requirements Modelling Extension

As indicated in Deployment Modelling Extensions, there was a need to extend CAMEL 2.5 in order to express and reference communication requirements. As such, the *RequirementModel* class in CAMEL has been extended in order to support the modelling of *CommunicationRequirement*s. A communication requirement can be considered as a hard constraint that does not have an explicit structure as in the case of resource and platform requirements. This is due to the fact that it can be expressed through the use of hierarchical feature models annotated with MDS concepts and properties. Further, as done also for these other kinds of requirements (i.e., resource and platform), a communication requirement is independent of the communication(s) it should be applied. As such, it does not reference any communication element. This has been done for reusability reasons by considering the fact that the same requirement could be attached to multiple component communications.

Please note that through a communication requirement we express constraints on potential communication quality attributes that are semantically annotated via MDS properties. In this way, we do not prescribe a particular way through which a platform that exploits CAMEL could enforce them. Thus, the platform could realise its own metrics for measuring the respective communication quality attribute that is constrained. On the other hand, we also cover the other alternative that a communication requirement could be expressed as an SLO. In that case, a specific communication metric will be involved (in the respective constraint) and would need to be supported either by the platform or the application itself.

Suppose that we need to express *CommReq1* communication requirement based on the example in the first section of the page Deployment Modelling Extensions. Further, suppose that we need a communication latency of less than 100ms and a jitter of less than 10ms. The respective communication requirement could be expressed in CAMEL 3.0 as follows:

```
requirement model ReqModel{
  communication requirement CommReq1{
    feature comQuality {
      [
MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.NetworkEntity.Net
workQoS ]
      attribute
latency     [MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.Netw
orkEntity.NetworkQoS.hasLatency] :
        , double 100.0) UnitTemplateCamelModel.UnitTemplateModel.MilliSeconds
      attribute jitter
[MetaDataModel.MELODICMetadataSchema.ApplicationPlacementModel.IaaS.NetworkEntity.Ne
tworkQoS.hasJitter] :
        , double 10.0) UnitTemplateCamelModel.UnitTemplateModel.MilliSeconds
    }
  }
}
```

As can be seen, this communication requirement has the *comQuality* feature which maps to *NetworkQoS* concept from MDS. This feature has two attributes that map to range constraints over *NetworkQoS* characteristics. The first attribute called *latency* expresses that network latency should be less than 100 ms. In this case, please see that this attribute is annotated with the *hasLatency* property of *NetworkQoS* concept from MDS and that it exploits the new feature of CAMEL 3.0 (see also Attribute Modelling Extensions) that allows expressing attribute range constraints (", double 10.0)"). Similarly, the second attribute is called *jitter* and expresses that network jitter should be less than 10 ms. In that case, please observe that this attribute is annotated with the *hasJitter* property of *NetworkQoS* concept from MDS and that it includes the ", double 10.0)" part that expresses the range constraint that jitter <= 10.0.

## d. Metric Modelling Extensions

### I. Object Context Extension

In order to support the modelling of metrics measuring the quality of communication, there is a need to map metric contexts to object contexts that can reference communications that areinvolved in the respective application (being modelled). To this end, a minor extension was performed in the *ObjectContext* class in order to refer also to *Communication* elements. In this respect, now an object context indicates that an object being measured can be an application component, a piece of data manipulated by the application or a communication (between application components).

In order to highlight this extension, we supply an example of a CAMEL model in which there is a definition of a metric measuring raw latency with respect to the communication between two application components:

```
camel model MyApp{
  deployment model MyDepModel{
    software A{
      required communication tomcatPortReq port 8080
    }

    software B{
      provided communication tomcatPortProv port 8080
    }

    communication A_to_B from A.tomcatPortReq to B.tomcatPortProv
    ...
  }

  metric model MetricModel{
    sensor Lat_Sensor{
      isPush
      config ''
    }

    raw metric RawLatency{
       template
MetricTemplateCamelModel.MetricTemplateModel.NetworkLatencyTemplate
    }

    object context AB_Lat{
       communication MyDepModel.A_to_B
    }

    raw metric context RawLatencyContext{
      metric RawLatency
      sensor MetricModel.Lat_Sensor
      object context AB_Lat
      schedule TenSec
    }

    schedule TenSec{
      interval 10
      time unit UnitTemplateCamelModel.UnitTemplateModel.Seconds
    }
  }
}
```

As can be seen from this example, we have an application with a deployment model that includes at least two components A and B, which communicate with each other based on port 8080. In the metric model of the application, we specify a raw metric (*RawLatency*) which refers to the *NetworkLatencyTemplate* from the metric template model in the (modelling) workspace. This template indicates that we measure *NetworkLatency* attribute, that the value type should be from (0, +inf), the unit of measurement is in seconds and the direction of values is negative (the smaller the better). We also define a push-based sensor (*Lat_Sensor*) that will provide the metric measurements along with an object context (*AB_Lat*) which indicates that the object being measured is the communication between components A and B. Further, we define a schedule (*TenSec*) indicating that the metric should be measured every 10 seconds. We combine all this information finally in the form of a raw metric context (which refers to the raw metric, the schedule, the object context and the sensor) named as *RawLatencyContext*. This metric context can now be utilised to construct SLOs or to model even more composite communication latency metrics like average latency.

REMARK: The MORPHEMIC platform does not yet support communication related metrics so such metrics need to be realised by the application itself.

## II. Metric Variable Extension

A metric variable is a construct that enables specifying utility functions, parts of utility functions as well as variables (and respective constraints) that can be incorporated in deployment reasoning models to be solved by the platform. Such a metric variable can be simple, predicating over the respective offer space of an application component, or composite, computed from other variables or metrics. As indicated in MORPHEMIC deliverable D2.3, composite metric variables can enable assessing either the supposed value of a quality metric or of the whole deployment solution utility through a particular formula when considering the current candidate solution produced by the Solver. However, such a formula does not cover precisely the mapping from the offering space to the metric space. Such a mapping is needed such that we can more precisely compute the value that a metric can have when the candidate deployment solution is applied. This problem is exaggerated when a utility function is computed from multiple metrics, which are not precisely computed through corresponding metric variables. The needed mapping could be either a precise formula expressing a function of the metric value from the underlying candidate solution components. Or it could map to the ability to predict the metric value based on the monitoring history of the application.

As such, it has been discussed and decided in the project that the second solution must be adopted. This means that metric variables should be able to directly refer to the metric that they “simulate”. In this respect, a metric variable can utilise its formula in order to make an imprecise prediction about the value that the metric will have based on the current deployment solution examined and, once enough metric measurements exist, then it can rely on the predicted value of the metric as being calculated by the platform itself. As such, the reference to the metric is an indication to the platform that it needs to predict the values of that metric, once this is possible.

For example, suppose that we need to assess the completion time (CT) for an application when examining each candidate solution during deployment reasoning. For this purpose, we will need to create a metric variable named *CT_var* and associate it with the following formula: *theta1* * *theta2* / (*c1* * *c2*) + *theta3*, where *theta{i}* are metrics: *theta1* represents the number of trainings left to do, *theta2* the percentile bound on task execution and *theta3* the elapsed time. *C1* and *c2* are metric (decision) variables with *c1* representing the number of instances and *c2* the number of cores. This formula intuitively enables selecting the more powerful application configurations when the application performance is bad and less powerful application configurations when the application performance is too high. Through the MORPHEMIC platform prediction feature and CAMEL’s respective extension, the metric variable *CT_var* can now be also associated with the context *CT_Context* of metric *CT* that measures the application’s completion time. As mentioned above, this is an indication that the *CT* metric measurements will be exploited for forecasting purposes when adequate historical input exists. The respective CAMEL metric model sub-part that shows the definition of the metric variable *CT_var* is given below:

```
variable CT_Var{
  template CompletionTimeTemplate
  formula ‘theta1 * theta2 / (c1 * c2) +
theta3’
```

```
  context CT_Context
}
```

## III. Window Extensions

### i. *Windows Processings*

The metric meta-model of CAMEL has been carefully designed in order to support the modelling of any kind of metric with all appropriate details covered like schedules, windows, object contexts and measurement formulas. It has been also inspired by Complex Event Processing (CEP) languages by also adopting some of their operators. However, this meta-model has a possible issue which has been unveiled based on the requirements of a MORPHEMIC use-case. In particular, when there is a need to perform a pre-processing of a measurement window before a composite metric value can be produced, CAMEL's metric meta-model regards this pre-processing as a new metric. As such, this creates two main problems: (a) the number of metrics in the overall metric hierarchy of the application gets increased and (b) such window pre-processing metrics do not map to real metrics but seem to be rather artificial, introduced in order to just address the respective pre-processing need.

Based on this issue and by studying the features of well-known CEP languages like Esper CEP language, it has been decided to extend CAMEL in order to more completely and elegantly address the aforementioned issue. To this end, a new concept has been introduced in CAMEL called *WindowProcessing* which represents a specific pre-processing that has to be performed over a measurement window. This concept covers mainly three kinds of pre-processing operators that exist in CEP languages:

- GROUP: the measurements in the window are grouped based on one or more criteria. The criteria can be the ones already supported in CAMEL (e.g., per instance, per host) or even custom ones.
- SORT: the measurements in the window are ranked based on ranking criteria (which could be the same as those used for the grouping of measurements in the previous operator).
- RANK: is an operator that combines the other two previous ones. First, the measurements are grouped according to a set of grouping criteria. Then, in each group, we retain only the last measurement. Finally, we rank the remaining measurements based on the ranking criteria.

A window can have multiple pre-processings which are applied in the order of their definition. The grouping/ranking criteria that are supported by CAMEL for accompanying the use of the above operators are the following:

- INSTANCE: the measurements are grouped based on the respective application component instance that they concern.
- HOST: the measurements are grouped based on the host of the respective application component instance that they concern. Thus, measurements originating from the same host are placed in the same group.
- ZONE: the measurements are grouped based on the (availability) zone to which they map.
- REGION: the measurements are grouped based on the region that they concern. Please note that one region can have multiple (availability) zones.
- CLOUD: the measurements are grouped based on the cloud that they concern.
- TIMESTAMP: this is logically speaking an exclusive ranking criterion as it is difficult and might not make sense to group measurements based on their timestamp.
- CUSTOM: this indicates a custom ranking/grouping criterion. In this case, the measurement being produced should have an information piece that maps to this criterion. Such an information piece usually takes the form of a key-value pair. Further, there is a need to provide the name of this criterion so as to match it with the key in the (key-value) pair (of the measurement) such that then the grouping or ranking can take place.

Please note that the first five from the above criteria types were already supported in CAMEL 2.5. In addition, a simple grouping of measurements can still be specified by relying on CAMEL 2.5 features due to forward compatibility reasons with respect to the CAMEL versions (2.5 and 3.0). However, on the other hand, when we need to specify other pre-processing operators or the combination of such operators, then we will resort to this new extension of CAMEL.

In order to showcase this part of the extension, we will attempt to cover some of the well-known cases (including the use of single operators or their combinations) that can be modelled in CEP languages. A detailed explanation of how all such cases can be covered by CAMEL 3.0 is given in MORPHEMIC deliverable *D1.3 Final Data, Cloud Application & Resource Modelling*.

Suppose first that we need to cover the CEP operator: rank(groupCriteria, topMeasurementNum, rankingCriteria). The semantics of this operator is that we group the measurements first based on the group criteria, then we retain the latest measurement in each group, next we rank the measurements based on the ranking criteria and finally we retain only a specific number of top measurements. Consider that the grouping criterion is HOST, the ranking criterion is TIMESTAMP and we retain only the 10 top measurements. The respective model part that we need to specify in CAMEL to support this is the following:

```
window ten_meas{
  type fixed
  size type measurements-only
  measurement size 10

  processing rankp{
    type rank
    grouping criterion host_cr{
      type host
    }
    ranking criterion ts_cr{
      type timestamp
    }
  }
}
```

As can be seen, we define a fixed window of 10 measurements size on which we apply a single processing of type *RANK*. The grouping in this processing relies on a single criterion called *host_cr* which is of type *HOST*. The ranking on this processing relies on a single criterion named *ts_cr* that is of type *TIMESTAMP*. As indicated above the semantics of this processing kind is to first group based on the grouping criteria (HOST), then to retain the latest measurement from the groups and then to sort based on the ranking criteria (TIMESTAMP). As the size of the window is 10, we actually achieve precisely the semantics of the rank (HOST, 10, TIMESTAMP) operator.

Suppose next that we need to apply the unique(groupCriteria) operator where the group criteria are HOST and CUSTOM and the name of the second criterion is ENVIRONMENT (i.e., whether we cover a production or development environment). The semantics of this CEP operator is that we group first the measurements based on the grouping criteria and then we retain the latest from each group. This operator can be covered in CAMEL 3.0 through the use of a GROUP processing and a sliding window of size 1. The respective way to model this is the following:

```
window unique_meas{
  type sliding
  size type measurements-only
  measurement size 1

  processing unique{
    type group
    grouping criterion host_cr{
      type host
    }
    grouping criterion cust_cr{
      type custom
      custom 'ENVIRONMENT'
    }
  }
}
```

As can be seen, we have created a sliding window of size 1 (*unique_meas*) and we applied to that window a GROUP processing (*unique*). In this processing, we have defined two criteria. The first, *host_cr*, has a *HOST* type. The second, *cust_cr*, is of type *CUSTOM*. As the type is *CUSTOM*, we need to define the name of the respective key ('ENVIRONMENT') in the measurements (to be grouped) through the *custom* attribute.

Lastly, suppose that we need to apply a combined CEP operator: groupwin(groupCriteria)#sort(topNum, sortCriteria). In this case, the semantics is that we first group based on a set of group criteria, then we sort each group according to a set of sort criteria and finally we retain in each group only the topNum measurements. Consider that we have the group criterion of CUSTOM (with ENVIRONMENT again as the key in the measurements), the sort criterion of TIMESTAMP and topNum is 10. This operator can be realised in CAMEL through a sliding window of size 10 where we first apply a group processing with CUSTOM criterion and then a sort processing with TIMESTAMP criterion. The respective CAMEL 3.0 model part is the following:

```
window unique_sort_top{
  type sliding
  size type measurements-only
  measurement size 10

  processing pgroup{
    type group
    grouping criterion cust_cr{
       type custom
       custom 'ENVIRONMENT'
    }
  }

  processing sgroup{
    type sort
    ranking criterion ts_cr{
      type timestamp
    }
  }
}
```

As can be seen, we have defined the *unique_sort_top* sliding window with a size of 10. In this window, we have introduced two processings. The first processing, named *pgroup*, is of type *GROUP*. It includes a single grouping criterion, named *cust_cr*, of type *CUSTOM* with the *custom* attribute (measurement key) having the value of *ENVIRONMENT*. The second processing, named *sgroup*, is of type *SORT*. It includes a single sorting criterion, named *ts_cr*, of type *TIMESTAMP*.

### *ii. New Window Size Types*

Further, we have extended the types of window sizes in order to include two new ones so as to enable CAMEL to cover additional window modelling cases:

- *TIME_ACCUM*: this corresponds to a special window that accumulates events until no event comes for a specific time period. This time period needs to be defined inside the window specification.
- *TIME_ORDER*: a (sliding) window that keeps measurements for a specific time period based on their arrival time and ranks them based on their timestamp. This is essential in case measurements come out of order as being published by an external/remote system to the platform. In this case, the time period for retaining the measurements needs to be supplied within the window definition.

In the following example, we define two windows, each mapping to one of these two new window types in CAMEL 3.0:

```
window time_accum{
  type fixed
  size type time-accum
  time size 10
  time unit UnitTemplateCamelModel.UnitTemplateModel.Seconds
}

window time_order{
  type sliding
  size type time-order
  time size 20
  time unit UnitTemplateCamelModel.UnitTemplateModel.Seconds
}
```

The first window is a TIME_ACCUM fixed window with a time period of 10 seconds. The second window is a TIME_ORDER sliding window with a time period of 20 seconds.

## IV. Complete Use Case

The following example has been drawn from a specific use-case of the project attributed to the ICON partner. It concerns the measurement of efficiency of a worker component. There is a need to compute both the raw and minimum efficiency of the workers. However, the minimum efficiency should be computed over a window that groups the measurements per HOST and then retains only one of the measurements in each group. Based on CAMEL 2.5, the measurement part of the application would require the use of 3 metrics instead of 2 as the pre-processing of the window would map to an intermediate metric between raw and minimum efficiency. Thanks to the new extension of CAMEL (CAMEL 3.0) related to window pre-processings, two metrics are only needed as now we have the ability to define window (pre-) processings for composite metrics. The following CAMEL model covers this by focusing mainly on the monitoring part (metric sub-model) and abstracting from details related to other parts (like the deployment one).

```
camel model windowProcess{
  application WindowProcessing{
  version '1.0'
  }

  deployment model WPDM{
    software Worker{
      script configuration config1{
        configure ''
      }
    }
  }

  metric model WPRM{
    measurable attribute Efficiency

    template EfficiencyTemplate{
      attribute Efficiency
      unit UnitTemplateCamelModel.UnitTemplateModel.Seconds
    }

    raw metric WorkerEfficiency{
      template EfficiencyTemplate
    }

    schedule MinEfficiencySched {
      interval 30
      time unit UnitTemplateCamelModel.UnitTemplateModel.Seconds
    }

    window MinWorkerWindow{
      type sliding
      size type measurements-only
      measurement size 1
      processing WorkerWinProc{
        type group
        grouping criterion Host{
          type host
        }
      }
    }

    composite metric MinWorkerEfficiency{
      template EfficiencyTemplate
      formula 'min(WorkerEfficiency)'
    }

    object context WorkerObjContext { component WPDM.Worker }

    composite metric context MinWorkEffContext{
      metric MinWorkerEfficiency
      schedule MinEfficiencySched
      window MinWorkerWindow
      object context WorkerObjContext
    }
  }
}
```

As can be seen above, the deployment model comprises the *Worker* application component with a script-based configuration. On the other hand, the metric model covers all the necessary details for measuring that component. In particular, it defines the non-functional attribute that is measured (Efficiency), a metric template (*EfficiencyTemplate*) associated with this attribute and the respective unit of measurement (*Seconds*). It further defines a raw metric called *RawEfficiency* that conforms to this template. Finally, it defines all necessary constructs for measuring the minimum efficiency of the *Worker* component:

- The *MinWorkerEfficiency* metric that also conforms to the *EfficiencyTemplate* and is computed based on a formula that applies the min function over the measurements of the *RawEfficiency* metric.
- A schedule called *MinEfficiencySched* covering a periodic time space of 30 seconds.
- A window called *MinWorkerWindow* which is sliding with a size of 1 and applies only one window pre-processing operator (*WorkerWinProc*) of type *GROUP*, based on a criterion (*Host*) with type *HOST*. This means that we group the measurements in the window based on the *HOST* criterion and then we retain only one from the measurements based on the size of the window.
- An object context called *WorkerObjContext* which refers to the object being measured, i.e., the *Worker* component.
- A composite metric context (*MinWorkEffContext*) that encompasses all the previous elements supplying all details needed to properly measure the min efficiency metric.

Please note that we have omitted from this example the sensor to be used for measuring the raw efficiency metric and the respective raw metric context as constitute details that are not necessary for the scope of this example, which is to showcase the use of window processing metrics in a real use-case.

### e. Metadata Modelling Extension

The MetaData Schema (MDS) is a conceptual vocabulary that enables enhancing CAMEL models in order to express various sorts of requirements through semantically annotated feature hierarchies. During application modelling CAMEL, MDS is utilised as another kind of CAMEL model (the metadata model in particular) that is included in the modeller's workspace so as to assist in annotating CAMEL models pertaining to complete applications. The metadata model itself is not modifiable by the CAMEL Designer, the graphical editor of CAMEL. This choice was made on purpose in order to discourage (application) modellers from modifying the metadata model. The main reason for this is that this model should be only modified by an administrator or a platform developer in case any kind of change is needed to reflect corresponding updates in the platform. As such, for the updating of this model, there is a dedicated editor, the MDS editor (ii. How to update the Melodic Metadata Schema (Yiannis)), to be used.

Based on the above analysis, it is crystal clear that MDS elements can be mapped to underlying platform features. In fact, as MDS is a very rich model, it includes multiple elements which are not always backed up by the MORPHEMIC platform. As such, it has been decided to modify CAMEL 2.5 so as to enable clearly indicating those MDS elements that are fully supported by the MORPHEMIC platform. This structural change was conducted on the metadata meta-model in CAMEL to incorporate the *implemented* boolean attribute in the definition of the MMSObject, i.e., the class that represents any kind of metadata element (like concepts, properties, and so on).

Based on this change, the MDS/metadata model in CAMEL can now clearly highlight the metadata elements that have been implemented in the MORPHEMIC platform. This enables the modeller to choose only those elements when required to semantically annotate CAMEL elements of his/her application. Further, in case the platform development evolves, any kind of implementation change could be immediately applied to the corresponding MDS element by modifying the value of its *implemented* attribute.

An example of the specification of this attribute is given below:

```
concept CPU {
  name "CPU"
  uri "mms:0e8c150b-9ba5-4161-8426-dcb77afb9144"
  description [ "This class refers to IaaS resources that use Central Processing
Units (CPUs) for carrying out software instructions that       specify the basic
arithmetic, logical, control and input/output (I/O) operations."
]
              implemented

  property hasManufacturer {
    name "hasManufacturer"
    uri "mms:0699935d-f5ec-4d9a-84db-8f7c6f2505d8"
    description [ "This property expresses as a string the manufacturer of the
certain processing unit." ]
    type data
    range uri "xsd:EString"
  }
  property hasNumberofCores {
    name "hasNumberofCores"
    uri "mms:a261c0d2-6a98-4c76-947c-9d4ef5456f26"
    description [ "This property denotes an integer that captures the number of CPU
cores available or requested." ]
    type data
    range uri "xsd:integer"
    implemented
  }

}
```

In this example, we can see that there is a concept named *CPU* representing a Central Processing Unit in a (virtual or physical) server. This concept is considered as implemented as it involves at least one implemented property. Indeed, by looking at this concept's properties, the *hasManufacturer* is not implemented (as this statement is missing and by default the property is then considered unimplemented) while the *hasNumberOfCores* is implemented. The latter property can be used to annotate attributes in features when expressing resource requirements and is taken into account by the MORPHEMIC platform in order to check the satisfaction of resource requirements for application components based on the VM/IaaS offerings provided by the supported cloud providers. As such, this property can be used in feature attributes while the *hasManufacturer* one should not, as it is not supported yet by the MORPHEMIC platform.

## f. CAMEL Configuration Alternatives

### I. Introduction

CAMEL is a rich, multi-DSL language that enables the specification of cross- & multi-cloud applications. Such applications comprise components that need to be deployed in the cloud. Each component might have a different form (e.g., micro-service, function, task) and different configurations, where the latter can be provider-independent or provider-specific. Configurations can also depend on the form of a specific component. For instance, micro-service components can be deployed through the use of operating-system-specific scripts or the use of container images while big data analytics components have big data processing framework-related configurations.

CAMEL is able to cover the modelling of all possible component configurations, either they are provider-specific or provider-independent. In this respect, it includes all the necessary concepts to represent each configuration kind. On the top-level, any configuration kind is a sub-class of the *Configuration* class. In a nutshell, the main configuration kinds supported by CAMEL are the following:

- *ScriptConfiguration*: covers the modelling of operating-system-specific configurations
- *ContainerConfiguration*: covers the modelling of container-based component configurations
- *ServerlessConfiguration*: covers the modelling of function/serverless component configurations
- *ImageConfiguration*: covers the modelling of FPGA-based component configurations
- *ClusterConfiguration*: covers the modelling of big-data-processing task configurations
- *PaaSConfiguration*: covers the PaaS-based configuration of a component

The mapping between component forms and the aforementioned configuration kinds can be seen in the following table:

*Table 3 CAMEL Component form and configuration mapping*

| Component Form | Configuration Kinds |
|---|---|
| Micro-Service | *ScriptConfiguration*, *ContainerConfiguration*, *PaaSConfiguration* |
| Function | *ServerlessConfiguration* |
| Task | *ClusterConfiguration* |
| FPGA-based | *ImageConfiguration* |

On the other hand, the way the above configurations map to specific resource types is depicted by the following table:

*Table 4 CAMEL Configuration and resource type mapping*

| Configuration Kind | Resource Kind |
|---|---|
| *ScriptConfiguration* | VM |
| *ContainerConfiguration* | Container |
| *ServerlessConfiguration* | (Specialised) Container |
| *ImageConfiguration* | VM with FPGAs |
| *ClusterConfiguration* | VM |
| *PaaSConfiguration* | VM |

To be noted that in the above table, we do not cover the case of resources/VMs with GPUs. Obviously, in this case, the fitting configuration kinds could be ScriptConfiguration, ContainerConfiguration & PaaSConfiguration.

In the following, we analyse how CAMEL supports each kind of configuration while we also supply an indication of whether a configuration of a specific kind can be provider-specific or provider-independent and in which particular cases. Examples from existing applications are given to cover all cases within the context of one configuration kind.

## II. ScriptConfiguration

A script configuration matches mainly micro-service-based components which need to be deployed on a certain VM. In this respect, it includes operating system (OS) commands (with respect to the OS of the VM) which enable to complete deploy such components. In fact, these commands cover the whole lifecycle of a component within a VM, such as downloading, installing, configuring, starting and stopping this component. Thus, in order to initiate the execution of a component, the first four commands are enough while the execution can be stopped (e.g., in case the component deployment is not needed any more) via the last command. While specifying this configuration, the user has also the opportunity to explicate the actual OS in which the commands will supposedly operate successfully.

A script configuration can be either (VM) image-dependent or not. In the first case, the id of the respective image needs to be specified. As images are usually provider-specific, this means that the configuration, in that case, will be provider-dependent. Thus, to support multi-cloud deployments, the user would need to supply image-specific configurations with respect to the same component for other cloud providers. On the other hand, if the image id is not specified, this means that the component can be deployed in any VM of any cloud provider. Thus, multi-cloud deployment is possible for that component without any need for additional configuration specifications.

There is also the scenario that the user might have some specific VM images that map to the same script configuration, each associated with a different cloud provider. In that case, the script configuration should not specify any image id but it needs to be associated with an image requirement, which would determine the ids of all those images for which the script configuration applies.

We should highlight that in CAMEL 2.5, a component is associated with a requirement set but in CAMEL 3.0, the configuration of a component is associated with a requirement set. In this document, we mainly focus on CAMEL 2.5. Thus, the supplied example models conform to the textual syntax of that version of CAMEL.

To be noted that when a script configuration is image-dependent, there is no need to explicate the respective OS. As images always have a fixed OS. While image-independence would require specifying also the OS for the respective configuration scripts to succeed.

Finally, please note that the commands in a script configuration could be also devops-tool (e.g., docker, puppet) dependent. This means that the platform would need to guarantee first the existence of the tool in the respective VM before the actual deployment of the corresponding component through the list of tool-specific commands. However, this is a scenario that is not currently covered by the Melodic platform (apart from Docker for which a specialised configuration kind has been already included).

### i. *First Example: Image-Independent Script Configuration*

This example has been taken from the PeopleFlow use-case of Melodic. It concerns the description of an HDFS component. This description includes a provider-independent script configuration while it is associated with a requirement for Ubuntu as the respective OS.

```
Software HDFSComponent {
    [MetaDataModel.MELODICMetadataSchema.Big_DataModel.DataManagement.DataStorage.Fi
leSystems.DistributedFileSystems.ClusterDistributed.CentralizedMetadata.HDFS]
            requirements HDFSRequirementSet
            required host HDFSHost
            provided communication HDFSProvidedPort port 8000
            script configuration HDFSConfig {
            download 'rm -rf ~/melodic && mkdir ~/melodic && cd ~/melodic &&
wget https://s3-eu-west-1.amazonaws.com/melodic.testing.data/APP.sh && chmod +x
~/melodic/APP.sh'
            install '~/melodic/APP.sh install'
            configure 'mkdir ~/test2'
            start 'printenv >> ~/melodic/env.txt && ~/melodic/APP.sh start'
            }
            longLived
 }
requirements HDFSRequirementSet{
            resource PeopleFlowRequirements.HDFSReqs
            os PeopleFlowRequirements.UbuntuReq
            horizontal scale PeopleFlowRequirements.HorizontalScaleHDFS
 }
```

### ii. *Second Example: Image-Dependent Script Configurations*

In this example, we supply the script configuration of the load balancer component of the FCR application. As it can be seen, there are 5 script commands specified: the 4 of them start the component when executed in order while the fifth is executed when the LB needs to be updated due to the increase or decrease of app component instances of the deployed FCR application. In addition, there is a specification of the image id from which the respective VM instance(s) of the load balancer component can be constructed.

```
software Component_LB {
```

```
            [MetaDataModel.MELODICMetadataSchema.UtilityNotions.UtilityRelatedProper
ties.Unmoveable]
            requirements LBRequirementSet
            required host WM_LBHostReq
            script configuration ComponentLBConfiguration{
                download 'sudo service tomcat7 stop && rm -rf ~/load_balancer &&
mkdir ~/load_balancer && cd ~/load_balancer && wget https://s3-eu-west-
1.amazonaws.com/xxx/yyy.sh && chmod +x ~/load_balancer/load_balancer.sh && printenv
>> ~/load_balancer/lb_download_env.txt'
                install 'printenv >> ~/load_balancer/lb_install_env.txt'
                configure 'printenv >> ~/load_balancer/lb_config_env.txt && export
PUBLIC_APP_IPS=($PUBLIC_ComponentPortAppReq) && sudo
~/load_balancer/load_balancer.sh install ${PUBLIC_APP_IPS[@]} && mkdir ~/test2'
                start 'printenv >> ~/load_balancer/lb_start_env.txt && sudo
~/load_balancer/load_balancer.sh start'
                update 'printenv >> ~/load_balancer/communication_lb_env.txt && sudo
~/load_balancer/load_balancer.sh configure $PUBLIC_ComponentPortAppReq && sudo
~/load_balancer/load_balancer.sh start'
            imageId 'xxx-ami-UBUNTU-16.04-AMD64-LINUX_YYY-ZZZ'
            }
            provided communication ComponentLBPort port 8087
            required communication ComponentPortAppReq port 8087
 }
```

### III. ContainerConfiguration

A component could be also deployed in the form of a container. In that case, the specification of the id of the container image has to be specified via which a respective container instance (hosting the desired component) can be created. Optionally, the specification can be enriched with additional details like the start and stop command for the container as well as a set of environment variables relevant for the container management tool to be exploited. Currently, the Melodic platform supports only Docker as the container management tool/framework. However, if extra tools/frameworks of this kind are supported in the near future, then the name of these tools would need to be also specified. This has been predicted in CAMEL through the attribute *containerType* that specifies the name of the container management tool.

Usually, container management tools like Docker are most commonly deployed in Linux OSs. In any case, the platform could freely choose to select the right OS for the VM that would host the container management tool as well as all containers managed by this tool. In this sense, there is no need for specifying a specific OS.

In the special case that a container needs to run in a very specific (hosting) environment (with respect to the actual VM), then the user would need to specify the image of the VM hosting that environment. In such a case, the main way this could be achieved would have been through the parallel association of the container configuration of the application component with an image requirement. In that case, supporting also true multi-cloud deployment, the user would need to specify all those image ids (which could belong to different providers) which could be used for creating VMs that can successfully host the respective container. This modelling way is suitable when all environments are similar and offer equivalent features to the container that is hosted. It is also suitable in case that environments are dissimilar or offer different features to the container. In that latter case, instead of having multiple image ids per image requirement mapping to one container configuration (for the same component), we would have just one.

### i. *First Example: Normal, Frequent Case with no Environment Dependencies*

This example has been taken from the TwoComponent application and has been modified to conform to CAMEL 2.5 (as originally container configurations were specified as script configurations). As it can be seen, what is actually needed here is the specification of the id of the container image as well as some docker arguments.

```
Software Component_DB{
            requirements DBRequirementSet
            container configuration MySqlDockerConfig{
                        container 'docker'
                        image 'mariadb'
                        feature environment
{                       [MetaDataModel.MELODICMetadataSchema.ApplicationPlacementMode
l.PaaS.Environment.Container.Docker.DockerArguments]
                                    attribute rootMySQLPassword: string
'MYSQL_ROOT_PASSWORD topsecret'
                                    attribute rootPassword: string 'ROOT_PW
topsecret'
                                    attribute Dbuser: string 'DB_USER melodic'
                                    attribute DbPassword: string 'DB_PASS
melodic123'
                                    attribute DbDatabase: string 'DB melodic'
                                    attribute mysqlPort: string 'port 3306:3306'
                                    attribute mysqlUser: string 'MYSQL_USER melodic'
                                    attribute mysqlPassword: string 'MYSQL_PASSWORD
melodic123'
                                    attribute mysqlDatabase: string 'MYSQL_DATABASE
melodic'
                        }
            }
            provided communication ComponentDBPort port 3306
}
```

### ii. *Second Example: Environment Dependencies-based Container Configuration*

We rely on the previous example which is identical in terms of the respective component configuration. What is actually changed is the inclusion of an image requirement in the set of requirements of the component at hand. This is shown in the following snippet:

```
deployment type model TwoComponentAppDeployment{
software Component_DB{
            requirements DBRequirementSet
            …
}
requirements DBRequirementSet{
            resource TwoComponentApp_Requirement.DBReqs
            horizontal scale
TwoComponentApp_Requirement.HorizontalScaleTwoComponentAppDB
            image TwoComponentApp_Requirement.Ubuntu18
}
}
requirement model TwoComponentApp_Requirement{
…
image requirement Ubuntu18 [ 'xxx','yyy' ]
}
```

## IV. ImageConfiguration

Such a configuration is relevant mainly in case of components that need hardware-accelerated resources like FPGAs. In this case, the component has to be supplied in a specialised form which encompasses the right environment to be deployed over a special-purpose VM. Usually, all special-purpose VMs can accommodate such an environment in the context of a certain cloud provider. As such, what is needed is mainly the URL of the image of the component and a requirement over the use of a specific provider. This modelling is proper in the sense that it can cover a future situation where multiple providers might support the same kind of FPGA-based (component) image. In any case, such a modelling highlights that image-based configurations are always provider-dependent/specific so multiple of them need to be specified to enable multi-cloud deployment. Optionally, the user could specify the type of accelerator, especially in case that other kinds of similar accelerators are to be supported by the MELODIC platform.

In the future case that specialised/custom images are allowed to access hardware-accelerated resources or FPGA-based components are tight to specific VMs/images, then the same modelling mechanism as in the case of container configurations could be exploited. In particular, an image configuration would need to be associated with an image requirement covering the ids of all images that can support its execution.

### *i. Example: Multiple, Provider-Specific Image Configurations*

As there was no use-case exploiting FPGA-based resources in Melodic, we rely on a fictitious component that performs a machine learning task. Such a component would have multiple image configurations, each mapping to a different cloud provider.

Please note that as we focus on CAMEL 2.5, there is no possibility to add a provider requirement to a configuration of any kind as requirements are associated with the application component level. Thus, in CAMEL 2.5, there is a *provider* attribute in an image configuration which is removed in CAMEL 3.0 to allow for the mapping of an image configuration to a requirement set that can include a provider requirement.

```
software ML{
            requirements MLRequirementSet
            image configuration ML_AWS{
                        image '.../xxx'
                        accelerator 'FPGA'
                        provider 'AWS'
            }
            image configuration ML_Alibaba{
```

```
                              image '.../xxx'
                              accelerator 'FPGA'
                              provider 'Alibaba'
              }
}
```

## V. ServerlessConfiguration

This kind of configuration is suitable for components that are deployed in the form of a function according to the serverless computing model. Such a configuration is usually provider-specific due to the heterogeneity in which each provider expects the binary or container form of a component to have. Currently, the Melodic platform expects that the binary or image of the component has been already constructed (e.g., through the use of external, provider-specific tools). As such, the user just needs to supply the URL from which the binary/(container) image of the component can be downloaded. Optionally, the user can supply some additional information, such as whether the deployment of the component can be continuous (currently not supported by MELODIC platform) and some particular configuration parameters of the respective hosting environment.

In case that the MELODIC platform is extended towards this direction, CAMEL enables to specify within a serverless configuration the way a component/function can be built. This can be accomplished by specifying a build configuration. Such a configuration includes details like the id of the artifact to be constructed, the URL of the component source code, the framework to be used for the build as well as optionally two strings covering the paths of the directories in the component's source code that can be included or excluded from the build, respectively.

Irrespectively of the way the binary/image of the component is supplied/produced, the user can specify an event configuration which covers two main configuration areas: (a) how the function can be built in terms of a specific URL and a specific HTTP method; (b) how frequently the function/component can be executed with default arguments in order to bypass the cold container problem. In the second case/area, the default arguments for the call should be specified through the feature scheduledExecutionConfig (in case the component/function requires them) while the schedule of the frequent calling has to be also modelled.

### *i. First Example: Provider-Specific Serverless Configuration*

The following snippet covers the description of a software component (dealing with the recognition of faces) in the context of an augmented reality application. Such a description includes the modelling of a serverless configuration that covers: (a) the download URL for the binary code of that component, (b) its event configuration in terms of its method name and type as well as (c) some environment-specific parameters which include the determination of the AWS Lambda handler and some application-related parameters.

### *ii. Second Example: Serverless Configuration with Build & Call Frequency Configuration*

We now present the same example which has been intentionally modified in order to include the determination of the way the binary code of the application component can be built from its source code (i.e., through the AWS SAM framework and the supply the URL from which the component source code can be downloaded).

```
Software RecognizeLambda{
            requirements RecognizeRequirements
            serverless
configuration recognize{
                        build configuration
RecognizeBuildConfig{
```

```
artifactId
"recognize"
                                    framework "sam"
                                    source code URL
"github.com/xxx/yyy"
                        }
                        event configuration
EventRecognizeConfig{
                                    method name
"face/recognize"
                                    method type post
                        }


                        config param attribute
RecognizeHandler
[MetaDataModel.MELODICMetadataSchema.ContextAwareSecurity
Model.Handler]: string "lambda_function.lambda_handler"
                        feature
RecognizeFunctionEnvironment
{                                    [MetaDataModel.MELOD
ICMetadataSchema.ApplicationPlacementModel.PaaS.Environme
nt]
                                    attribute
recognize_bucket : string "bucket xxx"
                                    attribute
recognize_faces : string "faces faces"
                                    attribute
recognize_file1: string "file1 xxx.dat"
                                    …
                        }
            }
}
```

## VI. ClusterConfiguration

Similarly to the case of a serverless configuration, a cluster configuration expects that the binary of a component already exists such that the user is required to specify only the URL from which to download this binary. In this case, though, the configuration may not be provider-specific but it is surely big data processing platform-specific. This indicates that this configuration is suitable in the case that the component takes the form of a (big data processing) task. This also signifies the need to also specify the actual big data processing platform that will manage the execution of the respective component. Optionally, the user can supply some additional details for such configuration kind, like what is the name of the class to execute (e.g., for Java components), what are the configuration parameters of the big data processing platform, what are the parameters to be passed to that platform and what are the parameters to be passed to the respective application component itself.

A cluster configuration presupposes the existence of the right environment to host the relevant big data processing framework client component. As the Melodic platform can create such a client component itself for those cloud providers supported, we have indicated above that there is no provider-dependence for this configuration kind. However, it can be the case that some additional software might need to be installed in the respective environment for the proper execution of the application component. Thus, such a software has to be within the environment on which both the big data processing client and the application component will be hosted. In such a case, there is a need for the user to prepare provider-specific images that already include the needed environment with the appropriate software stack. In that case, there will be a need to map the cluster configuration to an image requirement which can include the ids of all the images which convey the right deployment environment so as to support the application component's multi-cloud deployment.

*i. Example: Cluster Configuration for Spark*

```
software ComponentSparkWorker{
            requirements WorkerRequirementSet
            required host WM_WorkerHostReq
            cluster configuration
configSpark{                                       [MetaDataModel.MELODICMetadataSch
ema.Big_DataModel.DataManagement.BigDataProcessing.HybridProcessing.ApacheSPARK]
                        download URL 'https://s3-eu-west-
1.amazonaws.com/xxx/yyy.jar'
                        attribute className
[MetaDataModel.MELODICMetadataSchema.Big_DataModel.DataManagement.BigDataProcessing.
HybridProcessing.ApacheSPARK.ClassName]: string 'net.piliszek.mdfs.Main'
                        attribute sparkVersion: string '2.3.0' //todo: spark version
                        attribute appArguments
[MetaDataModel.MELODICMetadataSchema.Big_DataModel.DataManagement.BigDataProcessing.
HybridProcessing.ApacheSPARK.ApplicationArguments]: string '--data_bucket
melodic.testing.data/mdfs --aws_secret {{GENOMNEW_AWS_SECRET}} --aws_key
{{GENOMNEW_AWS_KEY}} --out_bucket melodic.testing.data/test-spark --decision_bucket
melodic.testing.data/mdfs --decision_file_name BIG_decision.csv --out_file_name
out.csv --data_file_name BIG_data.csv'
                        feature sparkArgs
{                          [MetaDataModel.MELODICMetadataSchema.Big_DataModel.DataMa
nagement.BigDataProcessing.HybridProcessing.ApacheSPARK.SPARKArguments]
                                    attribute executormemory: string 'executorMemory
6G'
                                    attribute drivermemory: string 'driverMemory 6G'
                        }
                        feature
sparkConfig{                                    [MetaDataModel.MELODICMetadataSchema
.Big_DataModel.DataManagement.BigDataProcessing.HybridProcessing.ApacheSPARK.SPARKCo
nfiguration]
                                    attribute conf: string
'spark.rpc.message.maxSize 512'
                        }
            }
}
requirements WorkerRequirementSet{
            resource Genom_Requirement.WorkerReqs
            horizontal scale Genom_Requirement.HorizontalScaleGenomWorker
            image Genom_Requirement.GenomImage
}
```

## VII. PaaSConfiguration

This configuration kind maps to the deployment of a component through a standardised PaaS API. Such an API then hides the details of how an IaaS service has to be managed and automates the respective deployment of a component without requiring the use of a multi-cloud application management platform like Melodic. In essence, what Melodic will have to do in that case would be just to exploit the API in order to achieve the management of the deployment of the respective component.

As a standardised API is exploited, we need to supply in this case the name of the API, its expected version and endpoint URL. In addition, we need to further provide the URL from which the environment specification for that component can be downloaded. That specification is a kind of component configuration for the PaaS API in the context of the current component at hand. Thus, obviously, this specification is PaaS API-specific.

Please note that currently such a configuration is not supported by the MELODIC platform.

### *i. Example: PaaS Configuration for a Micro-Service Component*

This example has been taken from the CloudSocket project (cloudsocket.eu) and has been adapted to conform to CAMEL 2.5. It concerns a CardDesigner application component that can be deployed through the use of a PaaS API.

```
software CardDesigner{
            requirements CardDesignerRequirementSet
            paas configuration CardDesignerPaaSConfig{
                        api 'PUL'
                        version '1.0'
                        endpoint 'https://xxx/yyy'
                        download URL ‘xxx2/yyy2’
            }
}
```