Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE Framework – Initial version Companion Document

Deliverable 1.5

Deliverable:	D1.5
Title:	DICE Framework – Initial version – Companion Documente
Editor(s):	Marc Gil (PRO)
Contributor(s):	Marc Gil (PRO), Ismael Torres (PRO), Christophe Joubert (PRO) Giuliano Casale (IMP), Darren Whigham (Flexi), Matej Artač (XLAB), Diego Pérez (Zar), Vasilis Papanikolaou (ATC), Francesco Marconi (PMI), Eugenio Gianniti(PMI), Marcelo M. Bersani (PMI), Daniel Pop (IEAT), Tatiana Ustinova (IMP), Gabriel Iuhasz (IEAT), Chen Li (IMP), Ioan Gragan (IEAT), Damian Andrew Tamburri (PMI), Jose Merseguer (Zar), Danilo Ardagna (PMI)
Reviewers:	Darren Whigham (Flexi), Matteo Rossi (PMI)
Type (R/P/DEC):	-
Version:	1.0
Date:	31-January-2017
Status:	First Version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

DICE partners

ATC:	Athens Technology Centre
FLEXI:	FlexiOPS
IEAT:	Institutul e-Austria Timisoara
IMP:	Imperial College of Science, Technology & Medicine
NETF:	Netfective Technology SA
PMI:	Politecnico di Milano
PRO:	Prodevelop SL
XLAB:	XLAB razvoj programske opreme in svetovanje d.o.o.
ZAR:	Universidad De Zaragoza

The DICE project (February 2015-January 2018) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869

Glossary

API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	ATL Transformation Language
CEP	Complex Event Processor
CI	Continuous Integration
CPU	Central Process Unit
CSS	Cascade Style Sheet
DDSM	DICE Deployment Specific Model
DIA	Data-Intensive Application
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DICER	DICE Rollout Tool
Dmon	DICE Monitoring
DPIM	DICE Platform Independent Model
DSL	Domain Specific Language
DTSM	DICE Technology Specific Model
EMF	Eclipse Modelling Framework
EPL	Eclipse Public License
FCO	Flexiant Cloud Orchestrator
GEF	Graphical Editing Framework
GIT	GIT Versioning Control System
GMF	Graphical Modelling Framework
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
IP	Internet Protocol
JDO	Java Data Objects
JDT	Java Development Tools
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MBD	Model Based Design
MDSD	Model Driven Software Development
MOF	Meta-Object Facility
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MW	MiddleWare
OCL	Object Constraint Language
OLAP	OnLine Analytical Processing
OMG	Object Management Group
OSGi	Open Services Gateway initiative

PDE	Plug-in Development Environment
PNML	Petri Net Markup Language
POM	Project Object Model (MAVEN)
QA	Quality-Assessment
QVT	Meta Object Facility (MOF) 2.0 Query/View/Transformation Standard
QVTO	QVT Operational Mappings language
RCP	Rich Client Platform
SCM	Source Code Management
SQL	Structured Query Language
SVN	Subversion Versioning Control System
SWT	Standard Widget Toolkit
TC	Trace Checking
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
UML	Unified Modelling Language
UML2RDB	Unified Modelling Language to Relational Data Base
URL	Uniform Resource Locator
UUID	Universal Unique IDentifier
VCS	Versioning Control System
VM	Virtual Machine
WST	Web Standard Tools
WTP	Web Tools Platform
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Table of Figures

Figure 1. Configure Simulation tool SSH connection 9
Figure 2. Launch the Simulation tool....9
Figure 3. Configure the Simulation tool execution.....10
Figure 4. Simulation tool result view.....10
Figure 5. Verification tool: modeling the uml model. 12
Figure 6. Verification tool: applying the stereotypes.....12
Figure 7. Verification tool: final diagram. 13
Figure 8. Verification tool: set value to the stereotype attributes. 14
Figure 9. Verification tool: run the configuration.....14
Figure 10. Verification tool: set the values to the configuration.....15
Figure 11. Verification tool: see the results. 15
Figure 12. Monitoring tool API. 17
Figure 13. Monitoring tool: Monitoring Service Visualization UI. 19
Figure 14. Delivery tool: web service properties configuration. 21
Figure 15. Delivery tool: menu entry tool execution. 21
Figure 16. Delivery tool: web service.....22
Figure 17. Delivery tool: upload the blueprint file (1).....23
Figure 18. Delivery tool: upload the blueprint file (2).....23
Figure 19. Delivery tool: progress of the deployment. 24
Figure 20. Delivery tool: deploying the blueprint.....24
Figure 21. Optimization tool: main web service.....27
Figure 22. Optimization tool: alternatives for public cloud analyses. 28
Figure 23. Optimization tool: selection of cloud admission control files. 28
Figure 24. Optimization tool: experiments. 29
Figure 25. DICER tool: configuring the model (1).....39
Figure 26. DICER tool: configuring the model (2).....40
Figure 27. DICER tool: configuring the model (3).....41
Figure 28. DICER tool: configuring the model (4).....42
Figure 29. DICER tool: configuring the model (5).....42
Figure 30. DICER tool: configuring the model (6).....43
Figure 31. DICER tool: configuring the model (7).....43
Figure 32. DICER tool: configuring the model (8).....44

Table of Contents

Glossary 3
Table of Figures 5
Table of Contents 6
1. Introduction 8
2. Simulation Tool 9
2.1. Configuration 9
2.2. Getting Started 9
3. Verification Tool 11
3.1. Installation 11
3.2. How to use D-VerT 11
3.3. DPIM Modeling 11
3.4. DTSM Modeling 11
4. Monitoring Tool 16
4.1. Installation 16
4.2. Getting Started 16
4.2.1. DICE Monitoring Service Administration 16
4.2.2. DICE Monitoring Service Visualization UI 18
5. Delivery Tool 20
5.1. Installation 20
5.2. Getting Started 21
6. Optimization Tool 25
6.1. Installation 25
6.2. Configuration 25
6.3. Getting started 27
7. Anomaly detection Tool 30
8. Trace Checking Tool 31
8.1. Installation 31
8.2. How to use DICE-TraCT 31
9. Enhancement Tool 33
9.1. Installation 33
9.2. Getting Started 33
10. Fault Injection Tool 35

11. Configuration Optimization Tool....36
11.1. Installation....36
11.2. Getting Started....37
12. Quality Testing Tool....38
13. Deployment Modelling (DICER) Tool....39
13.1. Introduction....39
13.2. DPIM Modeling....39
13.3. DTSM Modeling....40
13.4. DDSM Modelling....41
13.5. Running the DICER Tool....44
14. Conclusions....45

1. Introduction

This companion document to Deliverable 1.5 shows relevant information about each tool that is part of the DICE IDE, such as user installation guides, user guides and videos.

The complete information about the tools can be found in the deliverable for each tool. Also, for a global reference for the DICE Tools see the Github of the DICE Project.[1]

[1] https://github.com/dice-project

2. Simulation Tool

The DICE simulation tool is fully integrated with DICE IDE and comes already installed in it.

2.1. Configuration

The simulation tool requires some configuration from the user before being able to offer its functionality. The tool needs to access a Petri net analysis engine; hence it requires the hostname (or IP address), port number where it can be accessed, and credentials (username and password) of a user of the server hosting the Petri net simulator.

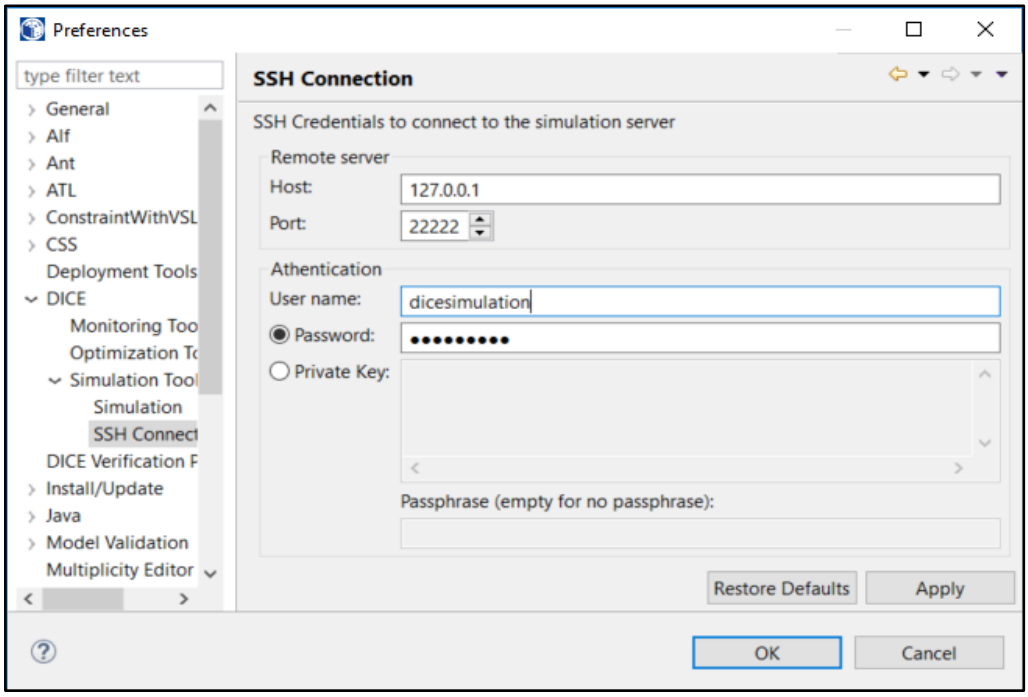

Figure 1. Configure Simulation tool SSH connection

After this configuration action, the tool is ready to simulate DIA DPIM and DTSM scenarios defined as profiled UML diagrams and offer the computed performance and reliability results to the user. The next paragraphs specify a simple scenario for the utilization of the simulation tool.

2.2. Getting Started

By clicking on the simulation tool icon when a DIA DPIM or DTSM scenario is selected in the IDE, the simulation tool User Interface is launched. Next figure depicts the position of the simulation tool launching icon in the IDE.

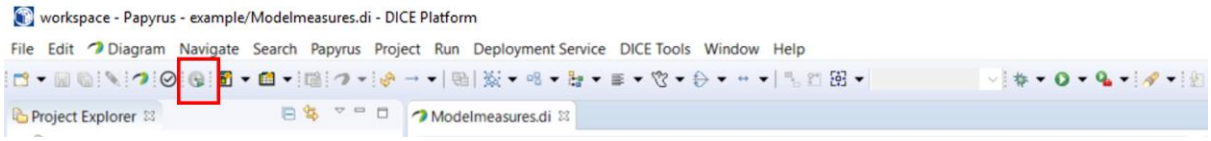

Figure 2. Launch the Simulation tool.

Through the GUI that opens when clicking in this button, the user can specify the metrics of interest that should be computed. The following figure shows a case where the user is interested has defined *Response Time, Throughput* and *Reliability* metrics in the profiled UML model whilst, for this concrete execution of the simulation tool, the user is not considering *Throughput* or *Reliability* measures and concentrates on the *Response Time* evaluation.

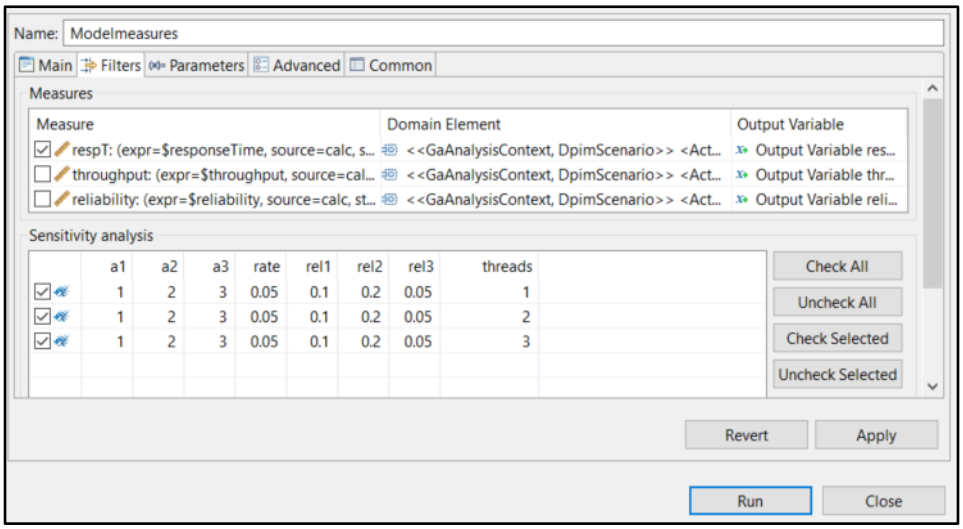

Figure 3. Configure the Simulation tool execution.

Then, the user clicks on "Run", the simulation process is launched, and results are returned as shown in the next figure. The uppermost part of the following figure shows that the simulation has finished correctly and, by double clicking on the result (for instance, in any point of the highlighted row in the figure), the computed results are opened. The lowermost part of the figure shows these computed results for the selected *Response time* metric. For instance, in figure the expected response time of the simulated DIA scenario is 5.90 s.

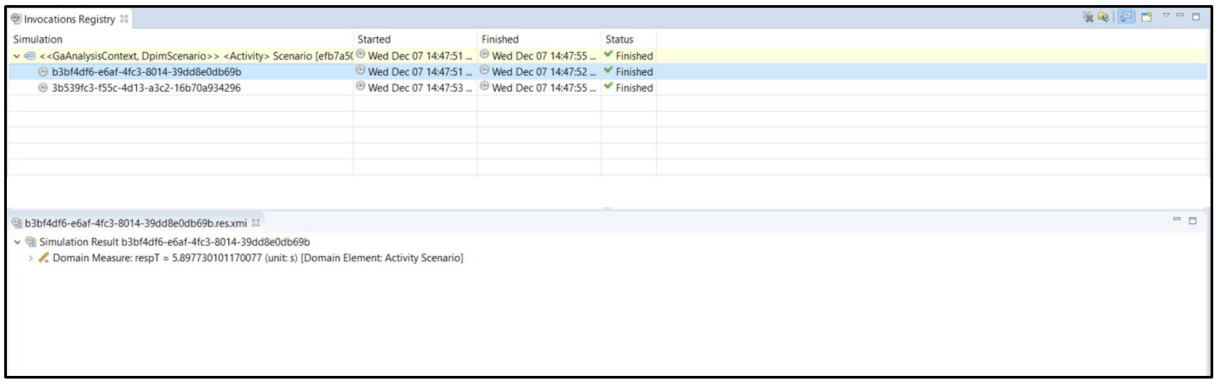

Figure 4. Simulation tool result view.

Further information on the capabilities and usage of the simulation tool is provided in D3.3[2], which details the intermediate version of the tool and is submitted contemporarily to this deliverable.

[2] D3.3 - DICE simulation tools - Intermediate version: http://www.dice-h2020.eu/deliverables/ . *To appear*

3. Verification Tool

D-VerT is the verification tool of DICE, which is fully integrated in the DICE platform. Currently, D-VerT supports the verification of Storm topologies that are defined at the DTSM level by means of annotated class diagrams. The instance of the verification problem is provided by the user through the DICE IDE which specifies (i) the DTSM diagram, (ii) the values of all the parameters needed to carry out verification and (iii) the computational nodes of the application that the user wants to analyse.

3.1. Installation

D-VerT can be installed automatically through the Eclipse installation service by linking the repository http://dice-project.github.io/DICE-Verification/updates in the "Install New Software" menu. Being a client-server application, D-VerT exploits an external service running on a server that carries out the mathematical verification. The server can be easily installed by running Docker that build the docker image available in the github repository.

3.2. How to use D-VerT

Storm topologies can be defined as DICE-Profiled UML class diagrams, in which each component of the application needs to be conveniently annotated. The class diagram is defined by simply dragging and dropping elements from the palette available in the DICE IDE.

The user starts the design of the DIA by creating a new Papyrus UML project and by selecting the "Class" kinds of diagram for his/her model.

Afterwards, the user opens the created class diagram and instantiates two packages, one for the DPIM model and another for the DTSM model and applies on the packages the DICE::DPIM and the DICE:DTSM UML profiles respectively. Specifically, as this guide exemplifies the creation of a Storm application, the user shall add to the project the "Core" and the "Storm" metamodels/profile that can be found in the DTSM entry.

3.3. DPIM Modeling

In the DPIM package, the user models the high level architecture of the DIA in a class diagram representing the computation of the application which elaborates input coming from possibly heterogeneous data sources. To this end, the user instantiates a new class and applies the <<DPIMComputationNode>> stereotype on it; he/she models the data sources, which can be either profiled by using the <<DPIMSourceNode>> of the <<DPIMStorageNode>> stereotypes, depending on the kind of data source; and, finally, he/she associates the computation nodes to the available data sources.

In this DPIM diagram the technological aspects of the application are not considered. These details are defined by means of the DTSM modeling.

3.4. DTSM Modeling

In the DTSM package, the user specifies which technologies implement the various components of the DIA. In particular, the user models the actual implementation of the computations declared in the DPIM along with all the technology-specific details.

For instance, let us suppose that the user wants to use Apache Storm to characterize the implementation of a <<DiceComputationNode>> appearing in the DPIM model. To this end, the user applies the stereotypes from the DICE::DTSM-Storm profile to describe the Storm application.

The following steps show how to define a Storm topology and to verify its model by means of D-VerT.

1. From the right hand-side panel, the user can drag-and-drop all the nodes defining the application into the design panel (Figure 5)

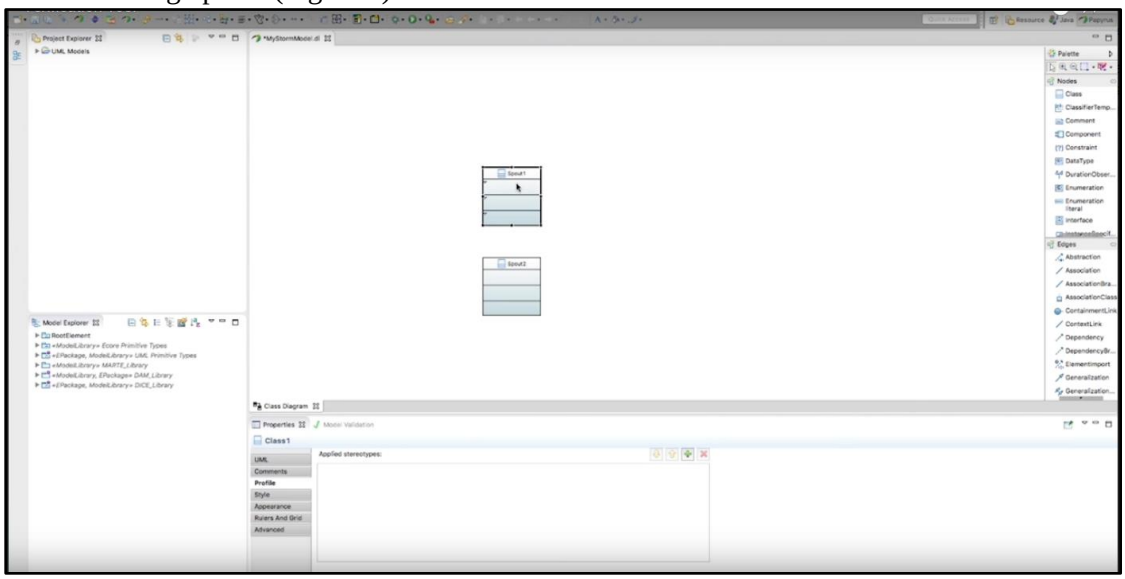

Figure 5. Verification tool: modeling the uml model.

2. From the bottom panel, the user can select the proper stereotype for each component of the application. The stereotype is chosen according to the kind of the node, that can be either a data source (StormSpout) or a computational node (StormBolt).

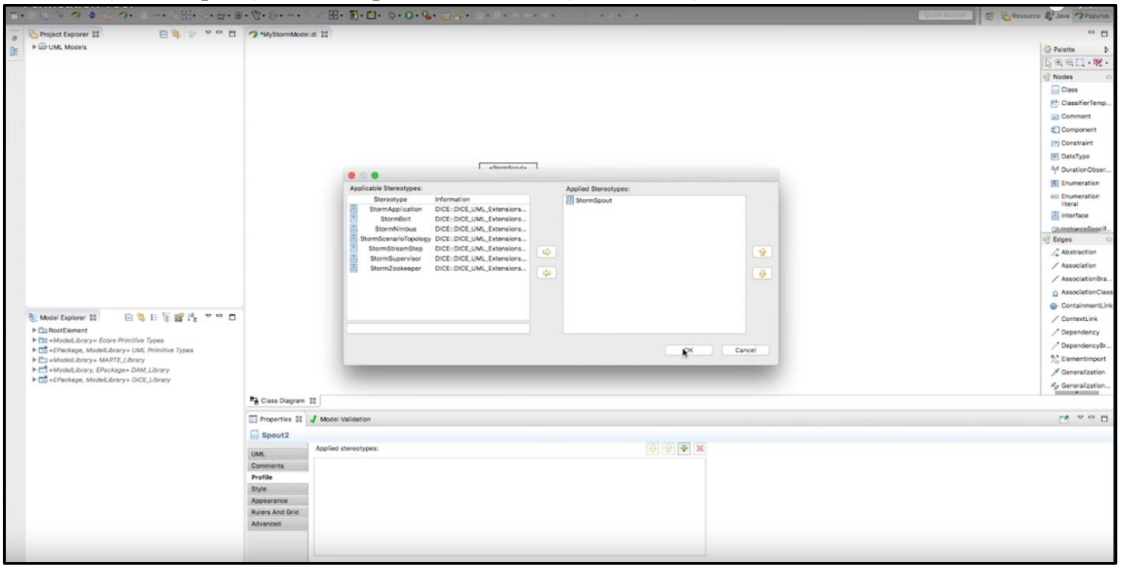

Figure 6. Verification tool: applying the stereotypes.

The user connects the nodes together through directed associations. Each association defines the subscription relation between two nodes: the subscriber, at the beginning of the arrow, and the subscribed node, at the end of the arrow.

4. Finally, the user should obtain a diagram similar to the one depicted in the following picture which will be verified with D-VerT.

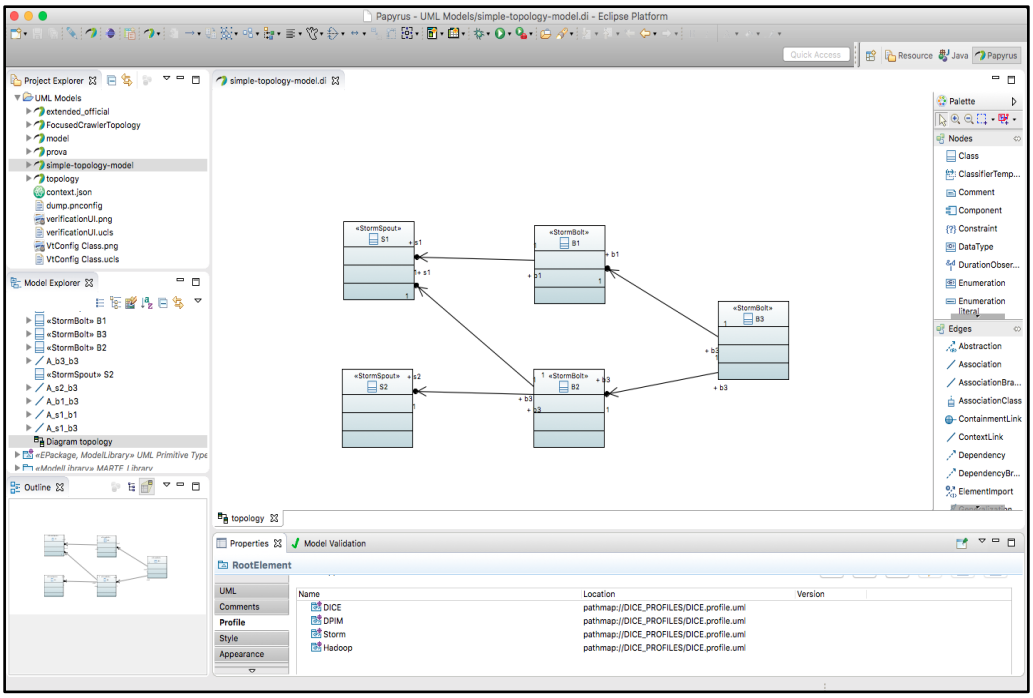

Figure 7. Verification tool: final diagram.

5. Before running the verification tool, the user specifies the values of the parameters related to the technology implementing the application. By selecting a node, the user can define, in the bottom panel, all the information needed for the verification. For instance, the timing features characterizing the spouts of the applications and the parallelism of the bolts. The values that are required to verify the topology are the following:
    a. parallelism, alpha, sigma for the bolts and
    b. parallelism, averageEmitRate for the spouts.

Figure 8. Verification tool: set value to the stereotype attributes.

6. Afterwards, the user can verify the topology with D-VerT. The Run configuration menu allows the user to define the best configuration for the verification task

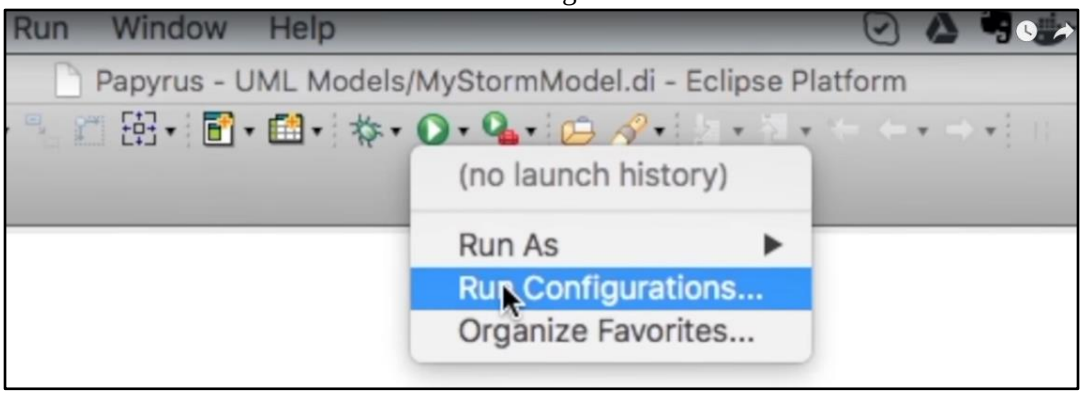

Figure 9. Verification tool: run the configuration.

7. The user provides the following information in the Run configuration window:
    a. The model to be verified (from the Browse menu).
    b. The number of time positions to be used in the verification process (time bound).
    c. The plugin that D-VerT uses to verify the model.
    d. The bolts that the user wants to test for undesired behaviors.

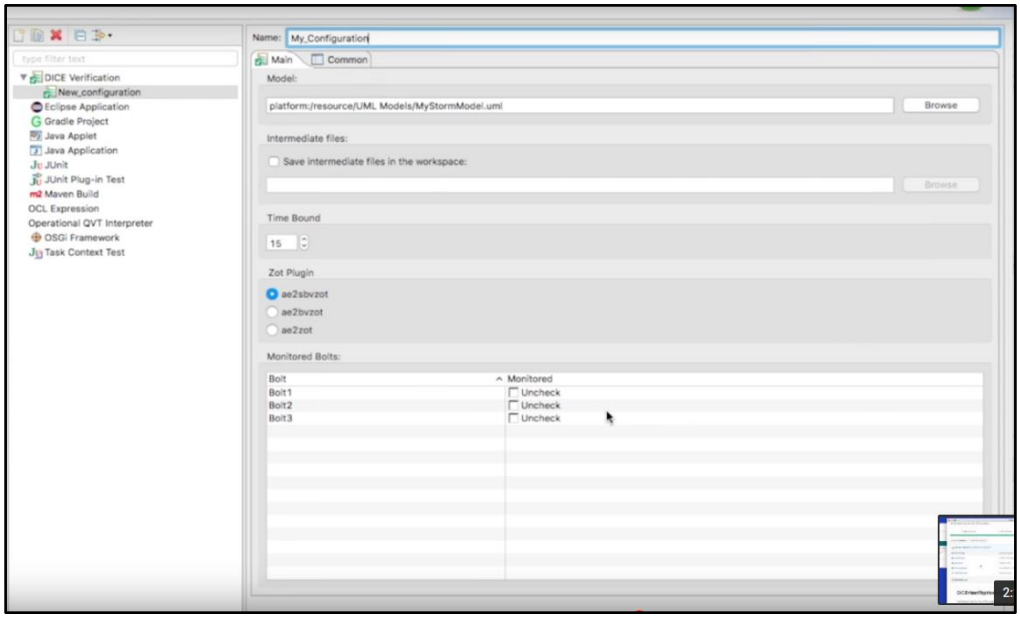

Figure 10. Verification tool: set the values to the configuration.

8. Now, the user executes D-VerT. In the D-VerT dashboard, the user can monitor the verification task running on the server.

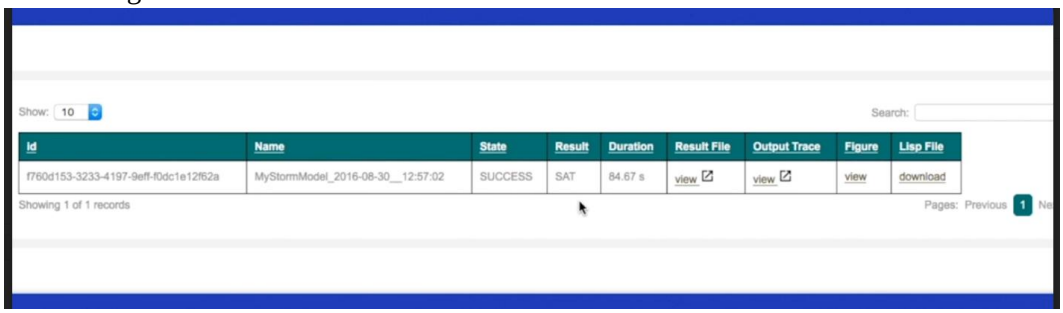

Figure 11. Verification tool: see the results.

9. The results after executing D-VerT are::
    a. The result of the verification, that is SAT, if anomalies are observed, or UNSAT.
    b. In case of SAT, the output trace produced by the model-checker shows the temporal evolution of all the model elements in detail and the graphical representation of the verification outcome shows the anomalies for a qualitative inspection.

4. Monitoring Tool

DICE monitoring platform[3] (DMon) collects, stores, indexes and visualizes monitoring data in real-time from applications running on Big Data frameworks. It supports DevOps professionals with iterative quality enhancements of the source code. Leveraging on Elastic.co's open source technology stack, DMon is a fully distributed, highly available and horizontally scalable platform. All the core components of the platform have been wrapped in microservices accessible through HTTP RESTful APIs for an easy control. DMon is able to monitor both the infrastructure (memory, CPU, disk, network etc.) and multiple Big Data frameworks, currently supported being Apache HDFS, YARN, Spark, Storm and MongoDB. Visualization of collected data is fully customizable and can be structured in multiple dashboards based on your needs, or tailored to specific roles in your organization, such as administrator, quality assurance engineer or software architect.

4.1. Installation

The DICE Monitoring Platform plug-in in Eclipse provides end-users with access to

- the platform's controller service REST API, the administration interface, and
- the visualization engine.

In order to configure the end-points for DICE Monitoring Service, do the following:

- select **Preferences** option from **Window** menu,
- under **DICE** entry, select **Monitoring Tools**,
- edit the preferences for monitoring platform
    - protocol: choose http or https to indicate which protocol the DICE monitoring service uses;
    - server: enter the IP or the hostname where the DICE Monitoring Service is accessible;
    - admin port: provide the port number of the DICE Monitoring Service's REST API
    - visualization port: provide the port number of the DICE Monitoring Service Visualization user interface
- click **Apply** to save preferences.

4.2. Getting Started

4.2.1. DICE Monitoring Service Administration

To connect to the DICE Monitoring Platform Administration interface, select **Open DICE Monitoring Service** under **DICE Tools** menu. This will open a new window of the built-in browser pointing to the DICE Monitoring Platform Controller service REST API. The address opened depends on the preferences provided in the installation step.

The opened view will show the Swagger UI for the DICE Monitoring Controller REST API, which exposes the specific operations for management and query of the platform, such as:

- control (deploy/start/stop) platform's core components (Elasticsearch, Kibana, Logstash)

[3] D4.1 - DICE Monitoring and data warehousing tools - Initial version and D4.2 - DICE Monitoring and data warehousing tools - Final version: http://www.dice-h2020.eu/deliverables/

- manage monitored nodes: register nodes, change configuration parameters and current status, deploy and configure node-level metrics, start/stop components
- query the platform, retrieve collected data defining metrics of interest, time window, etc.

To execute an operation against DICE Monitoring Platform, do the following:

1. Click on the verb (GET, POST, PUT, DELETE)
2. Fill in any details required by the operation
3. Press 'Try it out' button

This will issue the operation on the platform and result is sent back to the user.

For example, to list all nodes monitored by the platform, click GET button corresponding to */dmon/v1/observer/nodes*. In the response body received after 'Try it out!' button is pressed, you will get the list of monitored nodes.

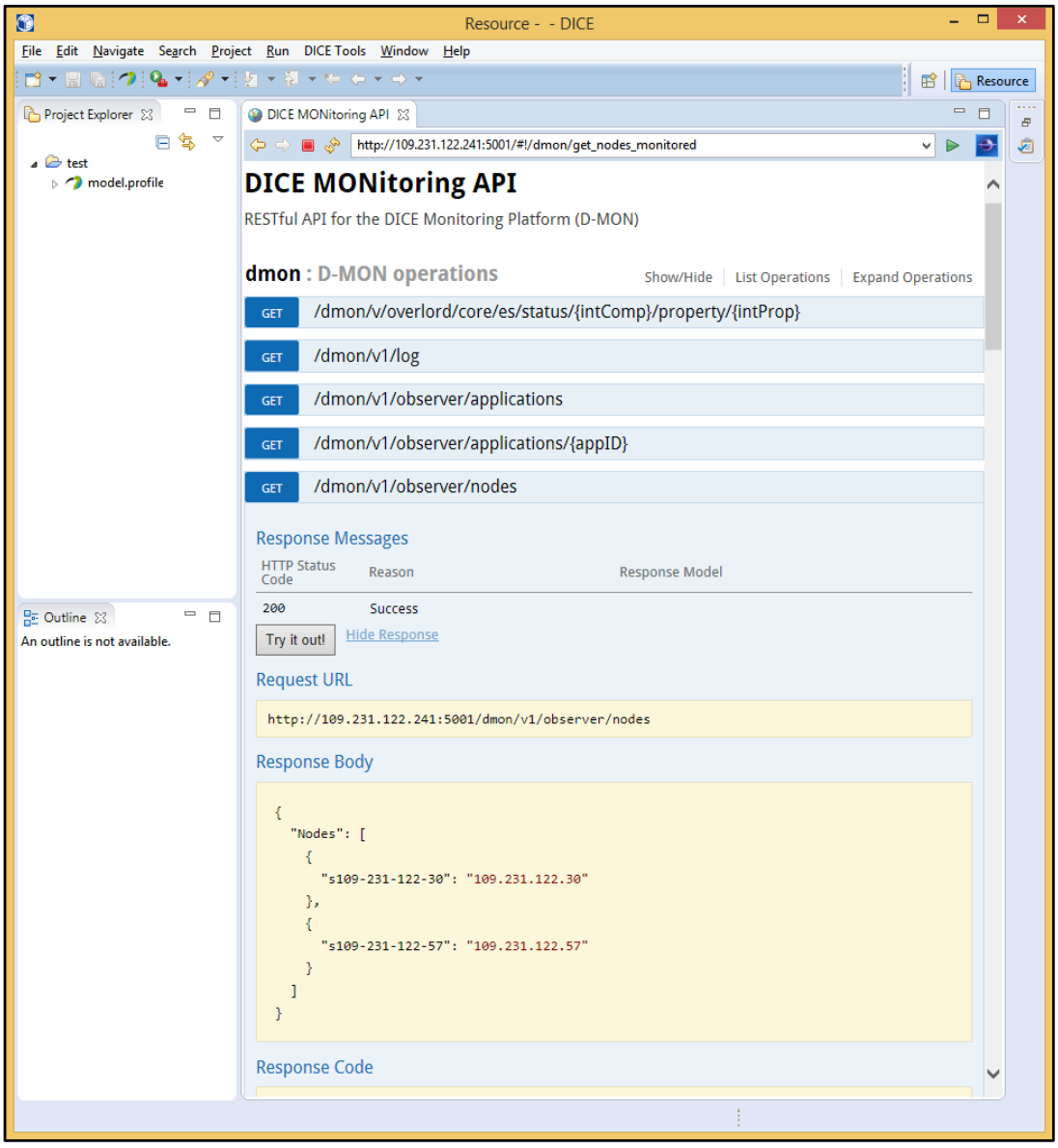

Figure 12. Monitoring tool API.

The operations exposed by DICE Monitoring Platform REST API are extensively described in several places:

- Deliverable D4.1 and D4.2 - Monitoring and data warehousing tools – Initial/final version[4], or
- DICE Monitoring Platform Wiki page[5], which is the up to date documentation and the authoritative source for it.

4.2.2. DICE Monitoring Service Visualization UI

To connect to the DICE Monitoring Platform Visualization interface, select **Open DICE Monitoring Service Visualization UI** under **DICE Tools** menu. This will open a new window of the built-in browser pointing to the DICE Monitoring Platform Visualization Engine UI. The address opened depends on the preferences provided in the installation step.

The opened view will show the Kibana user interface for DICE Monitoring service. The official Kibana Getting started website[6] introduces the end-user to Kibana experience. For example, to create a CPU load graph, do the following:

- from the main menu, select Visualize
- on 'Create new visualization' page select 'Line chart' and then select 'from a new search'
- select the appropriate index, that is logstash.*
- then, on metrics area (Y-axis), select Aggregation type = Average and Field = midterm
- repeat the step above for shortterm and longterm fields
- on buckets area (X-axis), select Aggregation type = Date histogram and Field = @timestamp; leave interval=Auto
- From top-right corner, select the time interval (last 15 minutes, last hour, today, last week, last month etc.)
- click 'Apply changes' button (the green play arrow)
- the right panel will contain the chart built appropriately.

[4] http://www.dice-h2020.eu/deliverables

[5] https://github.com/dice-project/DICE-Monitoring/wiki/Getting-Started

[6] https://www.elastic.co/guide/en/kibana/current/getting-started.html

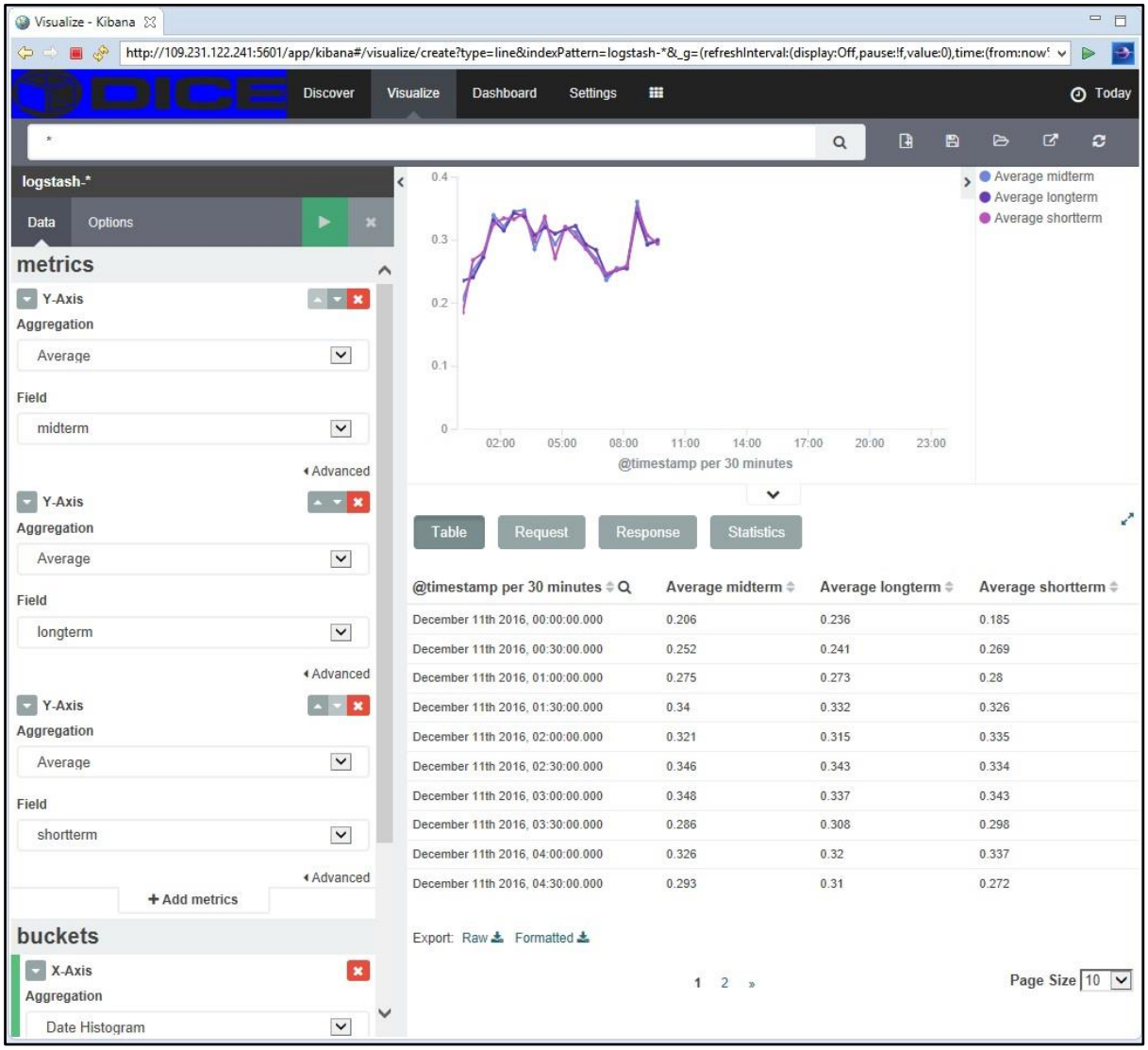

Figure 13. Monitoring tool: Monitoring Service Visualization UI.

5. Delivery Tool

DICE delivery tool[7] is a package for easy and fast deployment of DIAs. It enables the DevOps process for otherwise difficult processes such as installing and configuring Storm, Spark, Cassandra or any other Big Data services. The DICE delivery tool consists of the DICE Deployment Service[8] and DICE TOSCA technology library[9].

In this section we focus on using the DICE deployment tool in its stand-alone mode. This, of course, does not prevent the users from taking advantage of the interactions between DICE tools. For instance, one such important interaction is the use of the DICER tool in the IDE, which directly results in a TOSCA document used by the deployment tool.

The DICE deployment tool is also integrated with other tools outside the IDE, but the effects will be visible in the respective components' IDE plug-ins. In one scenario, the user can use the DICE deployment tool plug-in to deploy a monitored DIA, where the Deployment Service will automatically connect the relevant nodes with the DMon service. This will then be visible in the DMon's plug-in back in the IDE, letting the user monitor the runtime of the application.

5.1. Installation

The DICE delivery tool IDE plug-in is available as an open-source plug-in[10] and has an update site[11]. It also comes pre-installed with the DICE IDE.

Once the plug-in is installed, navigate to your DICE Deployment Service web page, or use the command line tool to create a deployment container for deploying your application. Please refer to the Container management section[12] of the DICE Deployment Service administration guide for instructions.

Next, configure the plug-in to use the container that you have just created.

1. In DICE IDE (Eclipse), open the Window menu and select Preferences.
2. On the left side of the Preferences dialog, click *Deployment Tools.
3. In the main part of the dialog, enter the appropriate values:
    - Protocol: choose http or https to indicate which protocol the DICE deployment service uses for both the Web GUI and the API.
    - Server: enter the IP or the host name where the DICE Deployment Service is accessible.
    - Port: provide the port number of the DICE Deployment Service's Web GUI and the API service.
    - Container: enter the UUID of the container created for deploying your application into.

[7] D5.2 - DICE delivery tools - Intermediate version

[8] DICE Deployment Service: https://github.com/dice-project/DICE-Deployment-Service/wiki

[9] DICE TOSCA library: https://github.com/dice-project/DICE-Deployment-Cloudify

[10] DICE Deployment IDE plug-in: https://github.com/dice-project/DICE-Deployment-IDE-Plugin/wiki

[11] DICE Deployment plug-in update site: http://dice-project.github.io/DICE-Deployment-IDE-Plugin/updates/

[12] https://github.com/dice-project/DICE-Deployment-Service/wiki/Installation#container-management

4. Click **OK** to confirm and save the preferences.

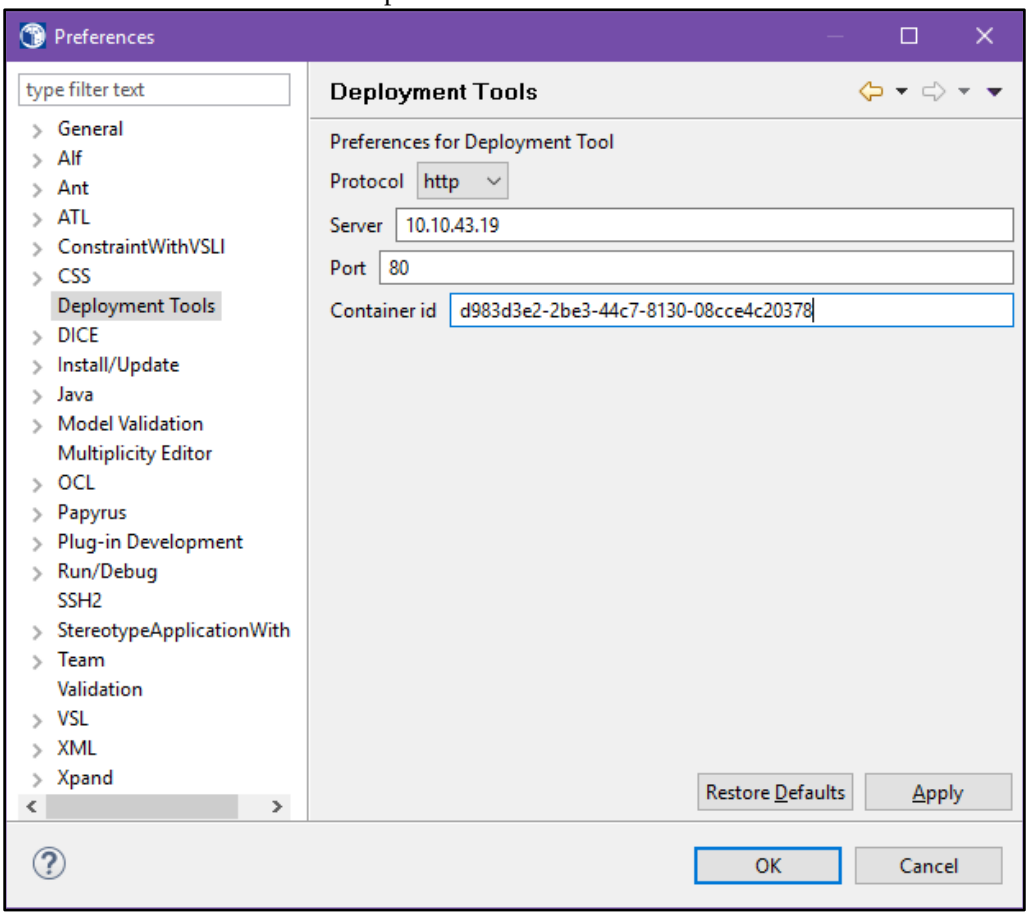

Figure 14. Delivery tool: web service properties configuration.

5.2. Getting Started

In the initial version, the DICE Deployment Service plug-in in Eclipse opens a Web user application view of the application's deployment container.

To start, click on the Deployment Service menu and select the DICE Deployments Tool UI option. This will open a new window of the built-in browser, pointing to the view of the container. The address visited depends on the preferences provided.

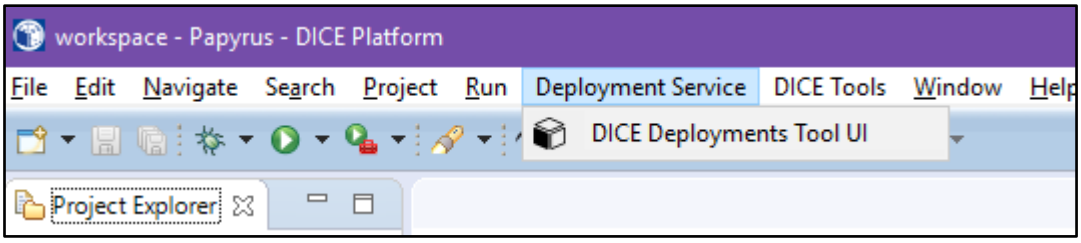

Figure 15. Delivery tool: menu entry tool execution.

The user interface will first challenge the user for the proper credentials.

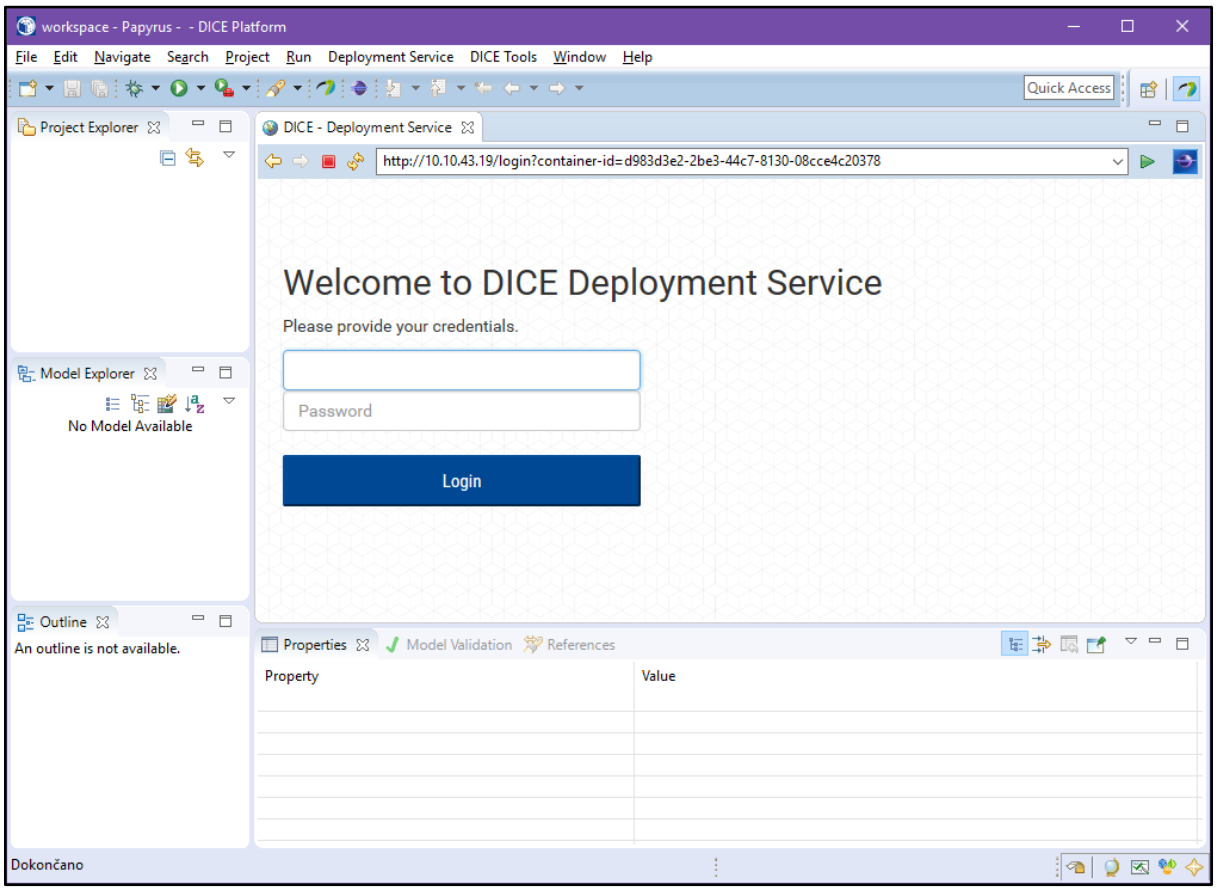

Figure 16. Delivery tool: web service.

Once you log in, you can upload a TOSCA blueprint of an application to be deployed. Simple blueprints are available in the example[13] folder of the DICE Deployment Service repository. Examples of Big Data application blueprints are also available[14].

1. Click on the Upload Blueprint button in the deployment container view.

[13] Simple blueprint examples: https://github.com/dice-project/DICE-Deployment-Service/tree/master/example

[14] DIA example blueprints: https://github.com/dice-project/DICE-Deployment-Examples

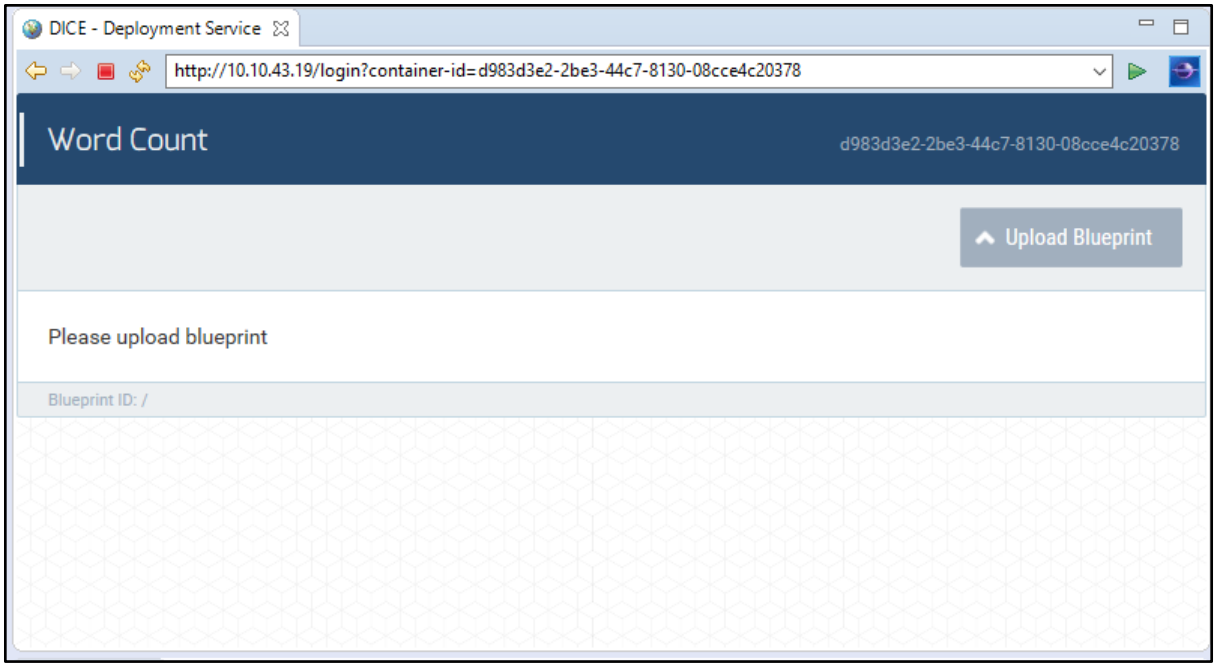

Figure 17. Delivery tool: upload the blueprint file (1).

2. Select a `.yaml` file containing the blueprint of the application.
3. Confirm the upload by clicking the Upload button.

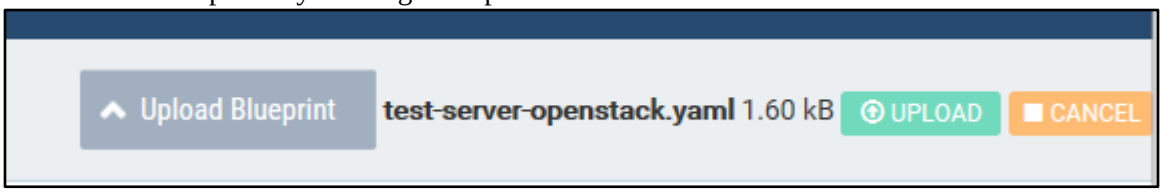

Figure 18. Delivery tool: upload the blueprint file (2).

The deployment process will then proceed unattended. The container view shows the progress of the deployment.

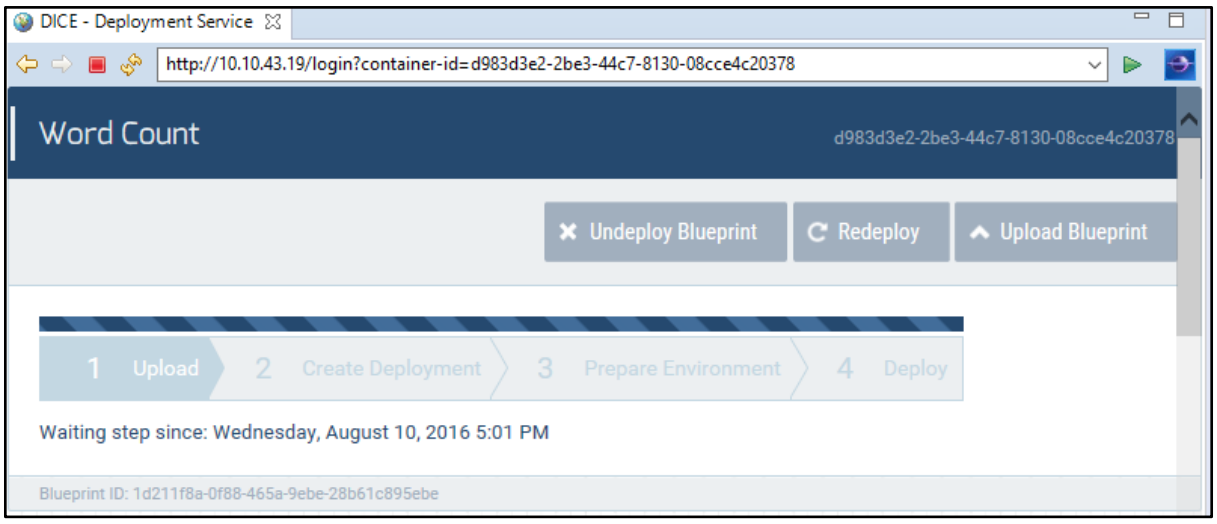

Figure 19. Delivery tool: progress of the deployment.

When the deployment is finished, a table showing output parameters of the application will appear. Normally, this shows relevant information on the deployed application, such as access point addresses.

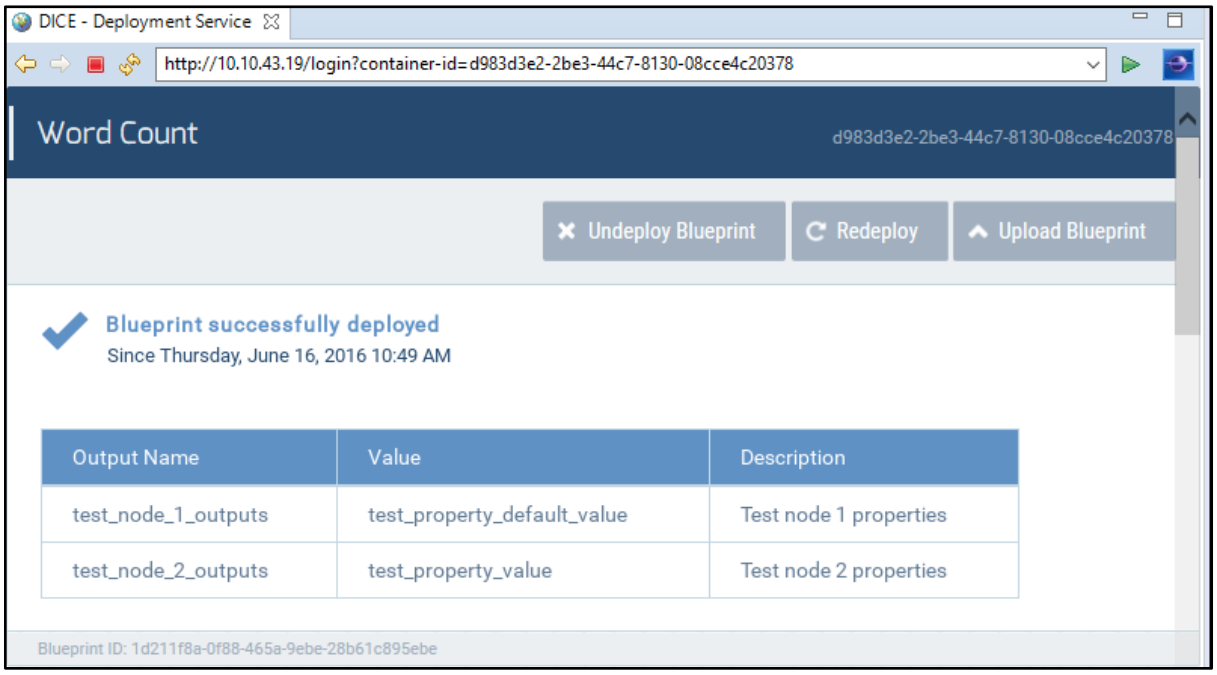

Figure 20. Delivery tool: deploying the blueprint.

To undeploy this application and replace it with a new one, the user needs to upload a new blueprint, repeating the steps listed above.

The **Undeploy** button tears down the application and frees the resources.

6. Optimization Tool

This sections reports installation instructions and a quick hands on example to get started with the DICE Optimization tool, D-SPACE4Cloud.

6.1. Installation

Build dependencies:

- Java 8 JDK
- Maven

Runtime dependencies:

- Java 8 JRE
- SSH
- AMPL
- Artelys Knitro or CPLEX
- JMT
- GreatSPN

In order to perform its optimisation procedures, the D-SPACE4Cloud back end relies on the third party solvers listed under "Runtime dependencies". Refer to the respective documentation for installation instructions. Also note that you can use either JMT or GreatSPN (or both), but you need to satisfy all the other runtime dependencies.

To download a D-SPACE4Cloud back end pre-built binary, look for the latest release in the Releases page of its repository[15]. There it is possible to find a relocatable jar file ready for download. The same goes for the D-SPACE4Cloud front end[16].

6.2. Configuration

The configuration files for both ends should be saved as `application.properties` in the same directory where the relocatable jar is stored and launched.

Below you can find an example configuration file for the back end. You should bear in mind that the solver paths (`{minlp,SPN,QN}.solver_path`) in the configuration file should be either command names available in the remote system `PATH` or absolute paths to the solver executables. The remote working directory (`{minlp,SPN,QN}.remote_work_dir`) must be a path where the remote use has full permissions. Moreover, the connection with solvers and simulators is established via SSH, hence you should provide an address (`{minlp,SPN,QN}.address`) and port (`{minlp,SPN,QN}.port`) where the remote SSH daemon listens, the path to your local known_hosts (`{minlp,SPN,QN}.known_hosts`) file, the remote user name (`{minlp,SPN,QN}.username`), and the path to an authorised private key file (`{minlp,SPN,QN}.private_key_file`). The accuracy and `significance` properties can be used to tune

[15] https://github.com/dice-project/DICE-Optimisation-Back-End

[16] https://github.com/dice-project/DICE-Optimisation-Front-End

the stopping criteria observed by the solvers. JMT allows to set a timeout for simulations, hence you can enforce a maximum simulation time in seconds via the `QN.max-duration` property. The `solver.type` property allows to choose the default simulator between JMT (`QNSolver`) and GreatSPN (`SPNSolver`). Depending on this choice, you might leave out the configuration of the unused simulator or even avoid installing it outright.

```
spring.profiles.active = test
solver.type = QNSolver
server.port = 8081
minlp.address = your.minlp.server.org
minlp.username = username
minlp.port = 22
minlp.remote-work-dir = /home/username/AMPL
minlp.ampl-directory = ampl
minlp.solver-path = knitroampl
minlp.known-hosts = ${HOME}/.ssh/known_hosts
minlp.private-key-file = ${HOME}/.ssh/id_rsa
minlp.force-clean = false
SPN.address = your.greatspn.server.org
SPN.username = username
SPN.port = 22
SPN.solver-path = swn_ord_sim
SPN.remote-work-dir = /home/username/GreatSPN
SPN.accuracy = 10
SPN.known-hosts = ${HOME}/.ssh/known_hosts
SPN.private-key-file = ${HOME}/.ssh/id_rsa
SPN.force-clean = false
QN.address = your.jmt.server.org
QN.username = username
QN.port = 22
QN.model = class-switch
QN.solver-path = /home/username/JavaModellingTools/JMT.jar
QN.remote-work-dir = /home/username/JMT
QN.accuracy = 10
QN.significance = 0.05
QN.known-hosts = ${HOME}/.ssh/known_hosts
QN.private-key-file = ${HOME}/.ssh/id_rsa
QN.force-clean = false
QN.max-duration = 7200
```

```
s4c.parallel = true
logging.file = log.txt
```

The following is an example configuration file for the front end. The ports listed in the front end `launcher.ports` property must be those configured in the back end configuration files as `server.port`. Currently you can use multiple back ends at once, provided they are reachable at the same `launcher.address`. Further, there is no need to deploy both the front and back ends on the same machine. `launcher.sol-instance-dir` allows to configure the directory where the front end stores the user uploaded files, whilst `launcher.result-dir` is the path where it stores the results retrieved from the back end.

```
launcher.sol-instance-dir = solInstances
launcher.result-dir = results
launcher.address = your.back.end.server.org
launcher.ports = 8081,8082
server.port = ${port:8080}
logging.file = logLauncher.txt
```

6.3. Getting started

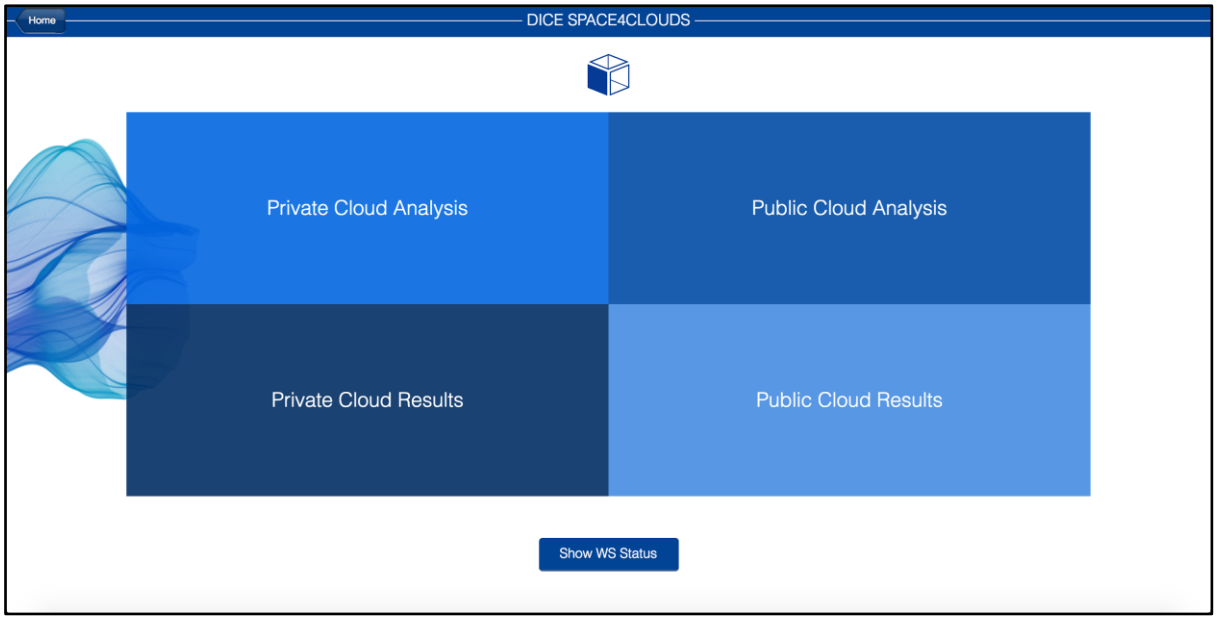

Figure 21. Optimization tool: main web service.

Launch your browser and type in the address where your front-end instance can be accessed. The figure above shows the home page where you will land. The small button at the bottom of the page shows the status of all the configured back ends. On the other hand, the four large tiles allow for data input to start the available analyses, in the upper row, and for results retrieval, in the lower row. The columns, instead, set apart private from public cloud scenarios. Clicking on either of the analysis tiles brings up a choice among possible alternatives, each provided with an explanation of the relevant scenario. The following figure shows, e.g., the alternatives available for public cloud analyses:

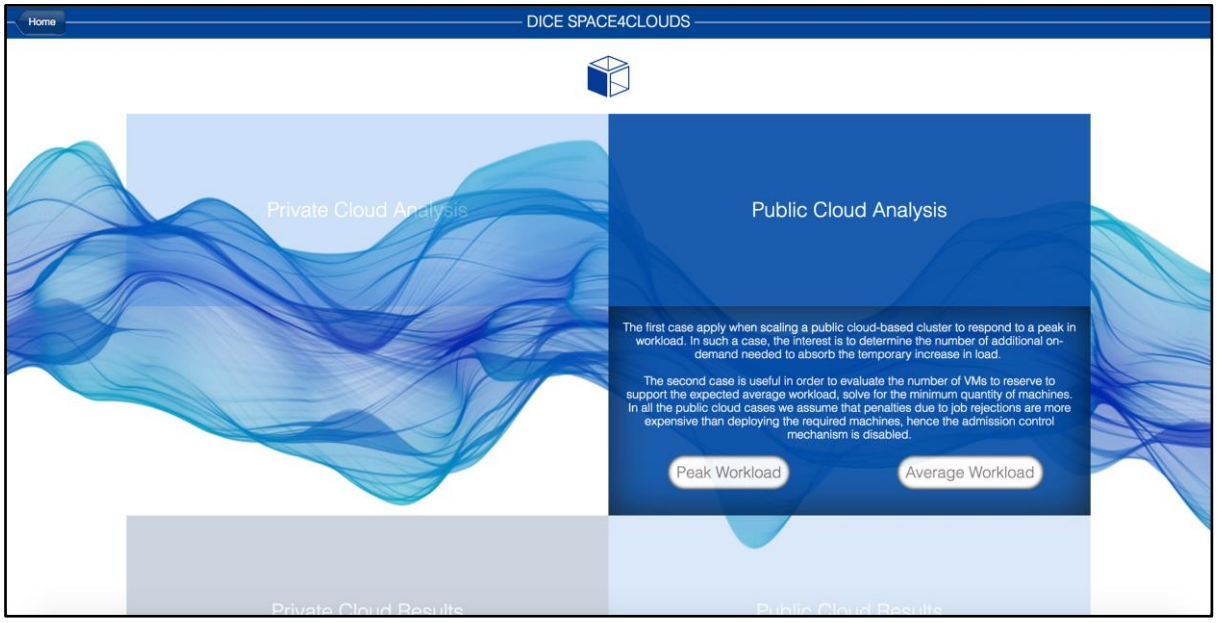

Figure 22. Optimization tool: alternatives for public cloud analyses.

After choosing the analysis of interest, you will be prompted to provide the needed data files. For an example, see the following figure. The dropdown list allows to choose between two alternative mathematical programming models to enforce the capacity constraint in private clouds. In the "Select a folder:" field you should, instead, point to the directory containing all the input files.

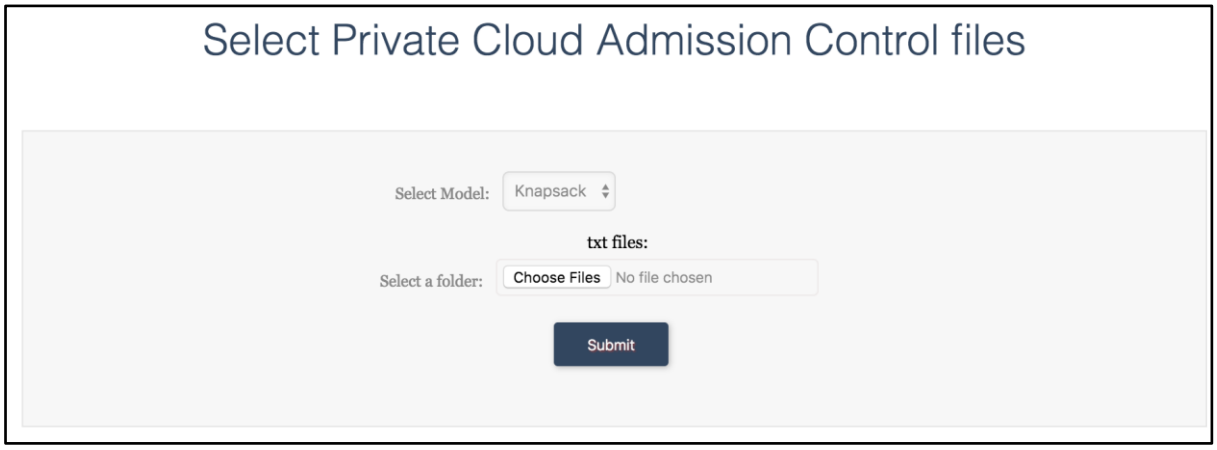

Figure 23. Optimization tool: selection of cloud admission control files.

As soon as the experiment is launched, the associated record populates the results page, shown below. Every record shows a unique identifier, date and time of submission, the configured accuracy, the overall number of runs involved and the number of completions, plus a status icon that highlights with a red signal experiments that went amiss. In addition to this data, there are icons to retrieve the input files and the results, as well as to discard a record or to restart an experiment.

Figure 24. Optimization tool: experiments.

7. Anomaly detection Tool

The Anomaly detection tool (AD) will reason on the base of black box and machine learning models constructed from the monitoring data. The predictive models built by the tool are then used to detect contextual anomalies in real-time monitoring data streams. A second use case supported by the Anomaly detection tool is the analysis of monitoring data produced during two subsequent executions of the same DIA. In this way, AD is able to discover anomalies introduced by latest code changes.

At M24, the tool is seamlessly integrated with the DICE Monitoring Service:

- in order for models to be able to detect not only point anomalies but also contextual anomalies, the tool will select a subset of data features from the Monitoring Service to train and validate a predictive model,
- the model is later stored in Monitoring Platform itself
- the anomalies detected are stored in the Monitoring Platform in a dedicated index.

The documentation about Anomaly detection tool can be found here[17], while the code is stored in Github[18]. More details about the design, modules and initial validation are available in D4.3 'Quality anomaly detection and trace checking tools - Initial Version' deliverable[19].

One module of anomaly detection tool deals with regression based checking of two consecutive runs of a DIA. The documentation for this module is available here[20], and video is available here[21].

Remark: Integration of anomaly detection tool in IDE is a work in progress due M30.

[17] https://github.com/dice-project/DICE-Anomaly-Detection-Tool/wiki

[18] https://github.com/dice-project/DICE-Anomaly-Detection-Tool

[19] http://www.dice-h2020.eu/deliverables/

[20] https://github.com/dice-project/DICE-Anomaly-Detection-Regression-Based-Tool/wiki

[21] https://www.youtube.com/watch?v=RnRxc09vphg

8. Trace Checking Tool

The Trace checking tool will be integrated in the IDE in future releases. In the current release, it cannot be launched from the IDE and it cannot connect to the D-mon platform. However, the tool can be used as a stand-alone application to validate Storm logs with a temporal property specified by the user.

This section summarizes the main steps that the user follows to run trace checking with the current implementation of the tool.

8.1. Installation

- Install Apache Spark (http://spark.apache.org/downloads.html) on your local machine
- Download the Soloist trace-checker from https://bitbucket.org/krle/mtlmapreduce/. Two options are available:
    - download MTLMapReduce.zip which provides a precompiled version MTLMapReduce.jar in folder /MTLMapReduce/scripts/
    - build the jar package following instructions therein

  Copy the jar package into the DICE-TraCT folder or make it available with a symbolic link, if it is stored elsewhere. The jar file must be called MTLMapReduce.jar
- Download the Python scripts from https://github.com/dice-project/DICE-Trace-Checking.
- Set environment variables JAVA_HOME and SPARK_HOME

```
export JAVA_HOME=/.../<your_Java_folder>
export SPARK_HOME=/.../<your_Spark_folder>
```

- Start Spark master

```
cd spark_folder
./sbin/start-all.sh
```

8.2. How to use DICE-TraCT

Trace checking is carried out by running **dicetract.py**. The interface to execute it is the following:

```
python dicetract.py
      -f <list of .log files>
      -t <json trace checking descriptor>
      -r <json regular expression descriptor>
```

dicetract.py has three distinct inputs that are defined through the command line options -f, -t and -r. To run trace checking the user specifies the log files of the Storm worker undergoing validation, a json file that describes the trace checking instance and a json file that provides the regular expression to be used for interpreting the log files. The property used for verification is defined in the trace checking descriptor and can be specified either by means of a given temporal formula or through the name of a user defined template.

dicetract.py runs a trace checking instance by means of a Spark job submitted to the Spark cluster through *spark-submit*.

The following command is an example of DICE-TraCT invocation; we assume that logs files w1.log and w2.log are stored in a local subfolder called logs/.

```
python dicetract.py -f ./logs/w1.log ./logs/w2.log -t tc.json -r re.
```

The integration of the Trace checking tool will be achieved through the following actions that will enable the use of the tool in Posidonia and Netf use cases.

1. In Posidonia, TC will be used to validate the behavior of the CEP given the log traces that it produces and the sequence of events it has elaborated. Therefore, TC integration requires (i) the definition of a format to exchange the traces and (ii) the implementation of a component providing the following functionalities: it allows the user to load a portion of a log from the Posidonia framework, to invoke the trace checker and to carry out the analysis of the log with respect to a given user property.
2. In Netf use case, TC will be used to validate DB logs with the aim of checking the validity of privacy restrictions that are enforced on some objects of the application (such as DB tables and columns of a given table) specified at DPIM/DTSM level. TC integration requires (i) the definition of a format to exchange the traces and (ii) the implementation of a component providing the following functionalities: it allows the user to
    a. to load a portion of a DB log from the DICE monitoring platform;
    b. to extract, from the DICE UML models of the application, a temporal logic formula that translates the privacy restriction modeled by the designer;
    c. to invoke the trace checker and to carry out the analysis of the log with respect to the formula obtained from the UML models.

In both the cases, the component to be implemented is an Eclipse plugin connected to the DICE IDE. It exploits an external service, running on a remote server, that will launch the trace checking engine given the instance obtained from the user through the IDE. The same architecture as the one implementing the verification tool D-VerT will be used for the trace checking tool.

9. Enhancement Tool

The DICE Enhancement tool will be externally integrated with DICE IDE. There are two modules of the Enhancement tool, Filling-the-Gap (FG) and Anti-Patterns & Refactoring (APR). More details are available in D4.5 - 'Iterative quality enhancement tools – Initial version' deliverable[22].

DICE-FG will be used to estimate and fit application parameter related to memory and execution times and annotate UML models. To perform the analysis, the DICE-FG first needs to obtain runtime information on the DIA from the DICE Monitoring Platform (DMon). To be specific, DICE-FG sends the JSON query string to DMon to collect the runtime data, DMon will return a JSON string which includes CPU utilization, job information, etc. This data is then used to automatically generate a valid set of input files for DICE-FG. Furthermore, DICE-FG also relies on the MATLAB Compiler Runtime (MCR R2016a) for execution. Therefore, it may be externally integrated in the IDE. DICE-FG focuses on statistical estimation of UML parameters used in simulation and optimization tool, and provides updated UML model for DICE-APR. DICE-APR will be used to detect anti-patterns of the performance model which is transformed from the UML model parameterized by DICE-FG. Therefore, it not only requires third party component - Epsilon to perform M2M transformation for further anti-pattern detection but also plans to using MCR for model refactoring.

Enhancement tool is due for final release at M30 and thus integration with DICE IDE will be completed during year 3. In order to install and run the Enhancement tool, the following instruction helps to guide users separately.

9.1. Installation

For DICE-FG, the installation is already specified in the GitHub repository[23]. Users can download the source code and run it on local machine according to the GitHub instruction.

For DICE-APR, it requires to install the Epsilon[24], which is a family of languages and tools for code generation, model-to-model transformation, model validation, comparison, migration and refactoring that work out of the box with EMF and other types of models, in DICE IDE.

9.2. Getting Started

The DICE-FG tool can be used as a stand-alone application. The DICE-FG tool requires runtime data via JSON from the DMon before being able to offer its functionality. It focuses on the specific input data that is requested to the user in order to use DICE-FG. The input parameters for DICE-FG tool are described in a dedicated configuration file (XML format) which specifies a complete DICE-FG analysis, consisting of a statistical distribution fitting step and an estimation analysis step. The instruction of getting started of the DICE-FG can be found in GitHub[25].

[22] http://www.dice-h2020.eu/deliverables/

[23] DICE-FG Installation: https://github.com/dice-project/DICE-Enhancement-FG/wiki/Installation

[24] Epsilon Update sites: http://www.eclipse.org/epsilon/download/

[25] Getting Started of DICE-FG: https://github.com/dice-project/DICE-Enhancement-FG/wiki/Getting-Started

At M24, the DICE-APR is still under development since this is planned for release at M30. The model transformation part (i.e. UML model (Activity Diagram and Deployment Diagram) annotated with DICE Profiles to LQN model) of the DICE-APR is tested in DICE IDE V0.1.3. The DICE-APR tool requires the updated UML model parameterized (e.g., host demand, schedule policy) by DICE-FG as input and performs the M2M transformations to obtain the corresponding performance model for further anti-patterns detection and refactoring.

10. Fault Injection Tool

The DICE Fault Injection Tool has been developed to generate faults within Virtual Machines and at the Cloud Provider Level. The purpose of the FIT is to allow cloud platform owners/Application VM owners a means to test the resiliency of a cloud installation as an application target. The FTI is a command line tool and as such can either be ran from a Java IDE such as Eclipse or packaged into a Executable Jar file to be ran. The current User/VM level faults implemented are:

- Shutdown random VM (Ignore tagged VM with "noshutdown" in random selection)
- High CPU for VM (Using Stress tool)
- High Memory usage for VM (Using Memtest tool)
- Block VM external access (Using ufw)
- High Bandwidth usage. (Using iperf, requires external iperf server ip to be passed)
- High Disk I/O usage (Using Bonnie ++)
- Stop service running on VM
- Shutdown random VM from whitelist provided by user (Note the whitelist does not check if VM exists or is a in a running state)
- Call YCSB on VM running MongoDB to begin workload test.
- Run JMeter plan

An example command using the executable jar:

```
java -jar DICEFIT --stressmem 2 512m ubuntu@111.222.333.444 -no c://SSHKEYS/VMkey.key
```

11. Configuration Optimization Tool

The Configuration Optimization tool is based on the Bayesian Optimization for Configuration Optimization (BO4CO) auto-tuning algorithm for Big Data applications. BO4CO is a command line tool, which we recommend to instantiate as a Jenkins job in order to periodically execute the configuration optimization.

Big data applications typically are developed with several technologies (e.g., Apache Storm, Hadoop, Spark, Cassandra) each of which has typically dozens of configurable parameters that should be carefully tuned in order to perform optimally. BO4CO helps end users of big data systems such as data scientists or SMEs to automatically tune the system.

11.1. Installation

BO4CO can be installed through Chef. We provide an automated installation of the BO4CO via a Chef cookbook. Detailed installation instructions are provided at https://github.com/dice-project/DICE-Configuration-BO4CO/blob/master/README.md and here summarized.

The BO4CO installation requires a Ubuntu environment (14.04) with the latest Chef Development Kit installed, available at https://packages.chef.io/stable. The installation cookbook is available under the consolidated DICE Chef Repository https://github.com/dice-project/DICE-Chef-Repository.git.

Before we run the installation, we just need to provide the configuration, pointing to the external services that the Configuration Optimization relies on. We provide this configuration in a JSON file. Here is an example (configuration-optimization.json):

```
{  "dice-h2020": {
    "conf-optim": {
      "ds-container": "4a7459f7-914e-4e83-ab40-b04fd1975542"
     },
    "deployment-service": {
      "url": "http://10.10.50.3:8000",
      "username": "admin",
      "password": "LetJustMeIn"
    },
    "d-mon": {
      "url": "http://10.10.50.20:5001"
    }
  }
}
```

Where the parameters have the following meaning:

- *ds-container* is the UUID of the DICE deployment service container dedicated to the application to run and optimize,
- *deployment-service* contains the access point and credentials of the DICE Deployment Service to be used by the CO,
- *d-mon* contains parameters used by the CO to connect to the DICE monitoring framework.

Now we can start chef

$ sudo chef-client -z -o recipe[apt::default],recipe[java::default],recipe[dice-h2020::deployment-service-tool],recipe[dice-h2020::conf-optim],recipe[storm-cluster::common] -j configuration-optimization.json

When the execution succeeds, the Configuration Optimization will be installed in */opt/co/* folder by default. The command will also install the Storm client (thanks to the recipe[storm-cluster::common] provided at the end of the runlist above).

11.2. Getting Started

A detailed getting started manual of the BO4CO tool is available at https://github.com/dice-project/DICE-Configuration-BO4CO/wiki/Getting-Started and here briefly summarized. The user of the tool needs to configure BO4CO by specifying the configuration parameters in expconfig.yaml. expconfig.yaml comprises several important parts: runexp specifies the experimental parameters, services comprises the details of the services which BO4CO uses, application is the details of the application, e.g., storm topology and the associated Java classes, and most importantly the details of the configuration parameters are specified in vars field.

For example, the following parameters specify the experimental budget (i.e., total number of iterations), the number of initial samples, the experimental time, polling interval and the interval time between each experimental iterations, all in milliseconds:

```
runexp:
  numIter: 100
  initialDesign: 10
  ...
  expTime: 300000
  metricPoll: 1000
  sleep_time: 10000
```

To run BO4CO you just need to execute the *run-bo4co.sh* bash script provided with the BO4CO distribution.

12. Quality Testing Tool

Quality Testing tool is scheduled for release in M24. The proposed integration plan for the tool is outlined below:

1. Quality Testing tool will be provided as a Java library in the **DICE IDE**, which a developer would need to integrate/interface with their application code.
2. The developer then would need to provide the following input to the Quality Testing tool in **DICE IDE**:
   a) The format of test load data;
   b) Load testing scenario (e.g. load volume, rate, test duration).
3. Next, the developer would need to deploy application in the 'testing mode' with the help of the DICE **Deployment tool**.
4. **Deployment tool** will pick up input parameters entered by developer and pass them to the **Quality Testing tool** during application deployment.
5. Test results can be visualised in **Jenkins**.

13. Deployment Modelling (DICER) Tool

13.1. Introduction

This section concisely describes the methodological steps to follow in order to create DICE UML models from scratch so that other DICE-provided tools can be run. In particular, just as a reference example, here we refer to the required modeling procedure to apply if you want to use the DICER tool.

For the scope of our example, a certain user is willing to use DICER, a tool developed in the context of DICE, along with the modeling procedure here presented, which enable, for this particular user, the automatic deployment of a Data Intensive Application (DIA), starting from its DICE UML models and generating TOSCA (Topology and Orchestration Specification for Cloud Applications) compliant deployment blueprints, which can, in turn, be processed by the DICE Deployment Service and deployed successfully.

In the following sections, you will be guided through the creation and definition within the DICE IDE of the various UML model envisioned by the DICE approach, which are the DICE Platform Independent Model (DPIM), the Dice Technology Specific Model (DTSM) and the Dice Deployment Specific Model (DDSM).

13.2. DPIM Modeling

Our imaginary user starts by creating a new Papyrus UML project and selecting the “Class” and “Deployment” kinds of diagram for our model.

Our user shall then open the created class diagram and instantiate two packages, one for his DPIM model and another for his DTSM model. On these two packages our user has to apply the DICE::DPIM and the DICE:DTSM UML profiles respectively.

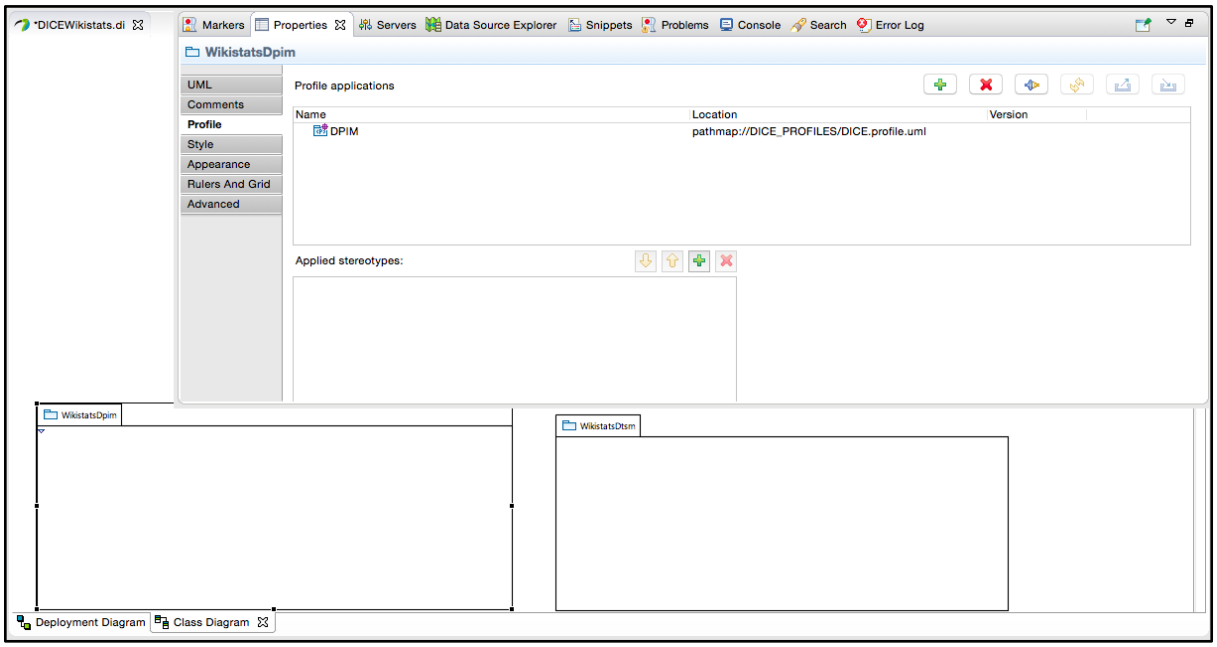

Figure 25. DICER tool: configuring the model (1).

At this point, in the DPIM package our user can model the high level architecture of his DIA, as a class diagram representing the kind of computations over various data sources. In order to model this, our user can instantiate a new class and apply on it the <<*DiceComputationNode*>> stereotype. Our user can also model various data sources which can be either profiled using the <<*DiceSourceNode*>> of the <<*DiceStorageNode*>> stereotypes, depending on the kind of data source. Our user can finally associate his computations to the available data sources.

Let us model a simple DIA called WikiStats, which basically analyze web pages from the popular Wikimedia website (a class with the <<*DpimSourceNode*>> stereotype) and stores the result of the analysis into a database (a class with the <<*DpimStorageNode*>> stereotype). So far we don't add any technological aspect of our application, such as the implementing technologies for the WikistatsApplication and WikistatsStorage nodes or the actual implementation of the WikistatsApplication node.

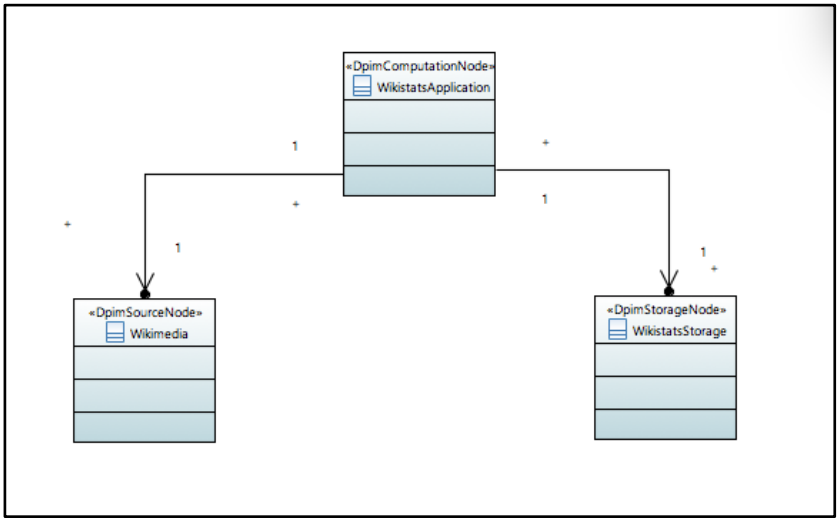

Figure 26. DICER tool: configuring the model (2).

13.3. DTSM Modeling

In the DTSM package you can model the actual implementation of the various computations you declared in the DPIM package, plus all the required technology-specific details. You can decide which technology to use for implementing the various components of our user DIA. For instance, let's suppose you want to use Apache Storm to detail the implementation of the "WikistatsApplication" <<*DiceComputationNode*>> in the DPIM model as a Storm topology. You can use the stereotypes from the DICE::DTSM-Storm profile to fully describe our user Storm application.

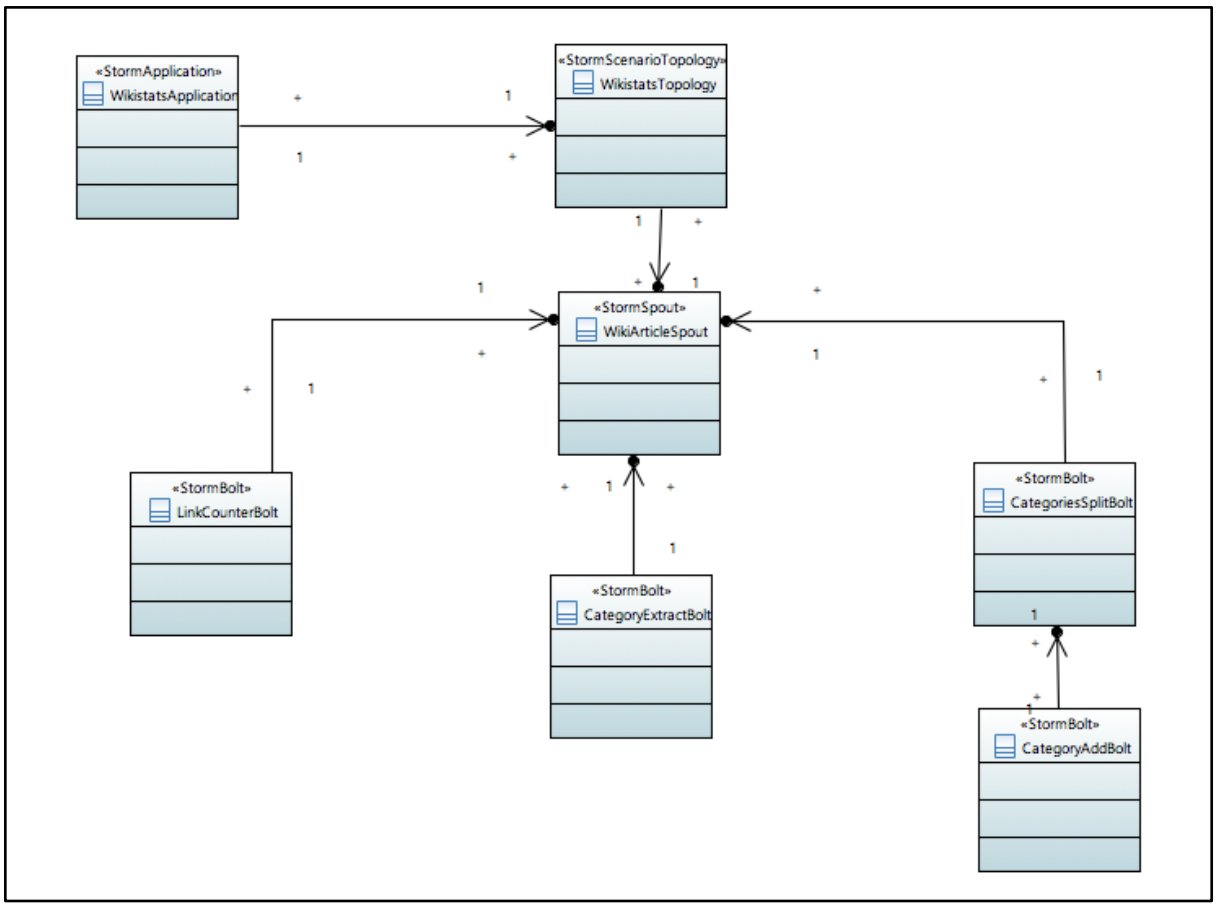

Figure 27. DICER tool: configuring the model (3).

Once you have such a DTSM diagram, you might execute some analysis on it, such as formal verification analysis to verify safety properties of our user model.

13.4. DDSM Modelling

As a last step our user can open the Deployment Diagram of our user UML model, to model the deployment of you DIA. You can start by applying the DICE::DDSM profile on our user Deployment Diagram. The available stereotypes allow you to model the required Cloud resources, such as virtual machines, and the allocation of the various Big Data middlewares required to execute our user application. For instance in the case of a Storm application, you need to instantiate two middlewares, the Apache Zookeeper platform and the Apache Storm platform, which depends on the former. Then you need to specify the required infrastructure for you middlewares. Let's suppose you want to deploy both Zookeeper and Storm on the same Cluster of VMs, which also means with the same number of replicas. In order to do so you can first instantiate a Device and apply the <<*DdsmVm*>> on it. Using the <<*DdsmVm*>> stereotype you can specify various properties of our user VMs cluster, such as the number of available VMs, and the Cluster provider.

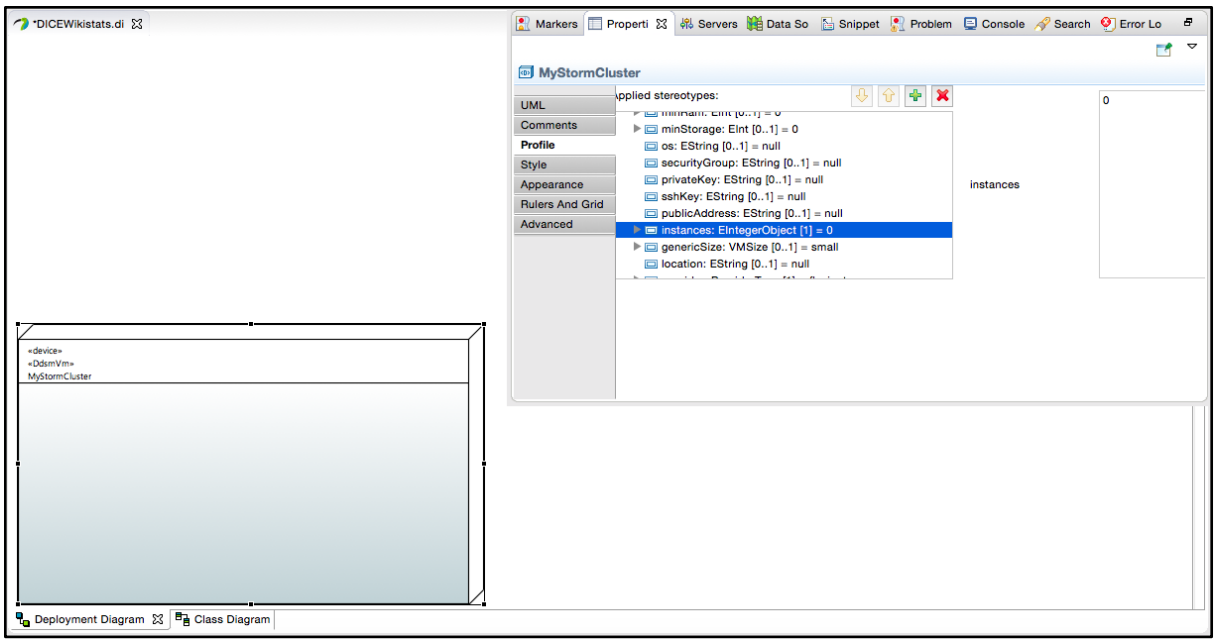

Figure 28. DICER tool: configuring the model (4).

You can then put inside our user VMs cluster a first Node and apply the <<*DdsmZookeeperServer*>> stereotype. Using the <<*DdsmZookeeperServer*>> stereotype you can specify the various properties of our user Zookeeper cluster. You can repeat the same process for a second Node tagged with the <<*DdsmStormCluster*>> stereotype and you can model the dependency of Storm on Zookeeper by simply connect the two nodes with a Dependency element.

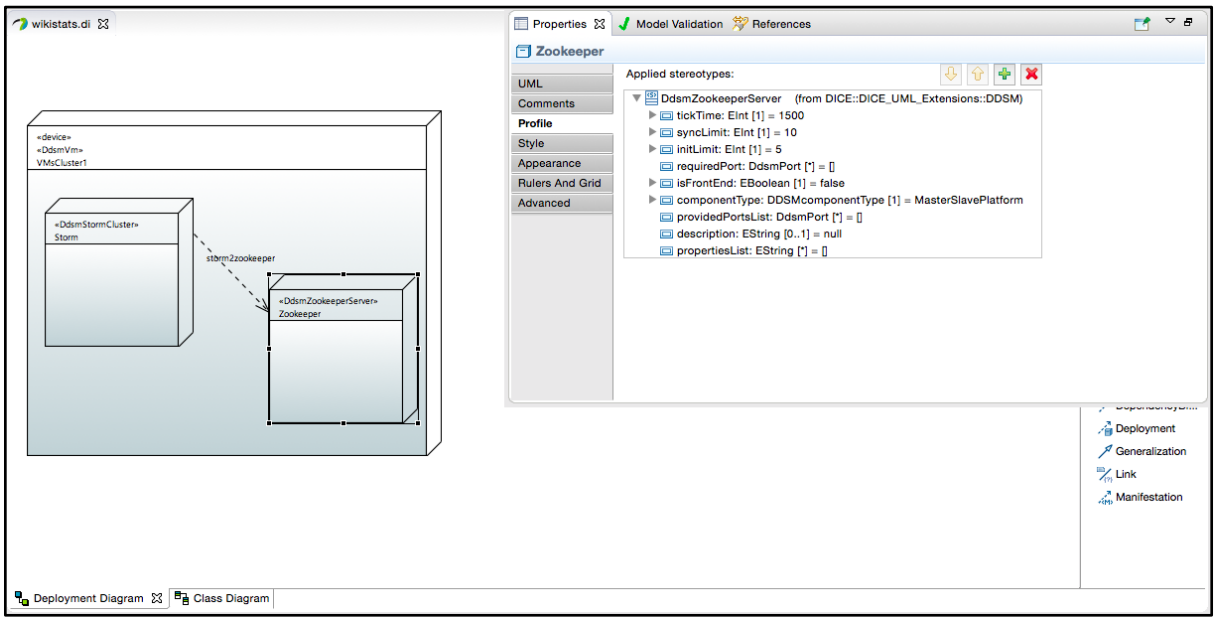

Figure 29. DICER tool: configuring the model (5).

Since our Wikistats application will use store the result of its analysis into a database, let's imagine the user decide to use Apache Cassandra for this purpose. Thus we have to model the deployment of a Cassandra cluster, which will be accessed by our application. Let' also imagine we want to host the Cassandra cluster on a different VMs cluster from the one hosting Storm and Zookeeper. We then instantiate a new Device and we apply the <<*DdsmVm*>> stereotype on it. We can then put a new Node within the just created Device and apply the <<*DdsmCassandraCluster*>> stereotype on it. We can finally specify the various properties of our Cassandra cluster using to the applied stereotypes.

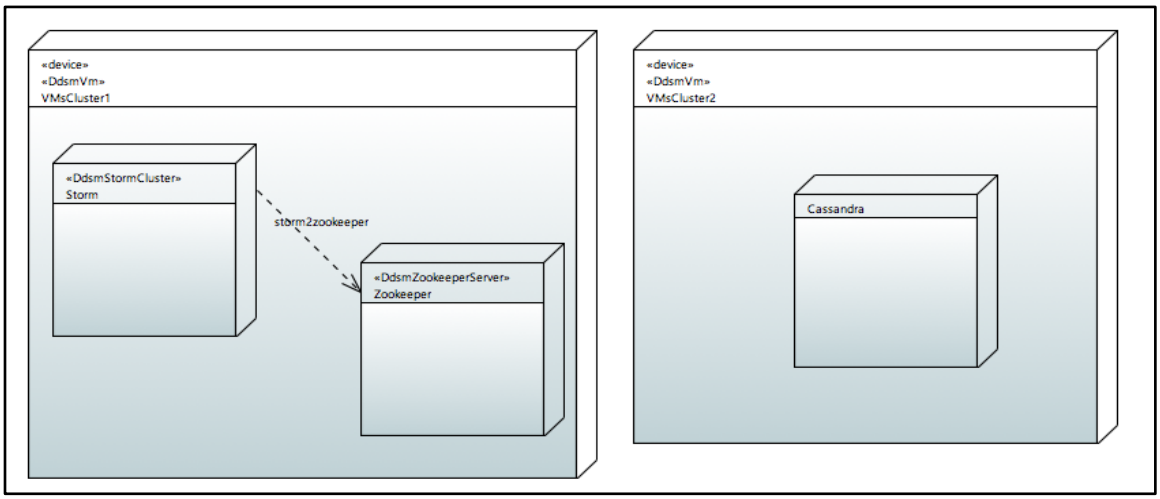

Figure 30. DICER tool: configuring the model (6).

Finally, our user can model the deployment of his application by instantiating an Artifact on which to apply the BigDataJob stereotype. Our user can specify the required deployment information for his application, such as the location of the application runnable artifact and thus specify to which of the available Big Data middleware our user DIA has to be submitted, using the DICE Deployment service.

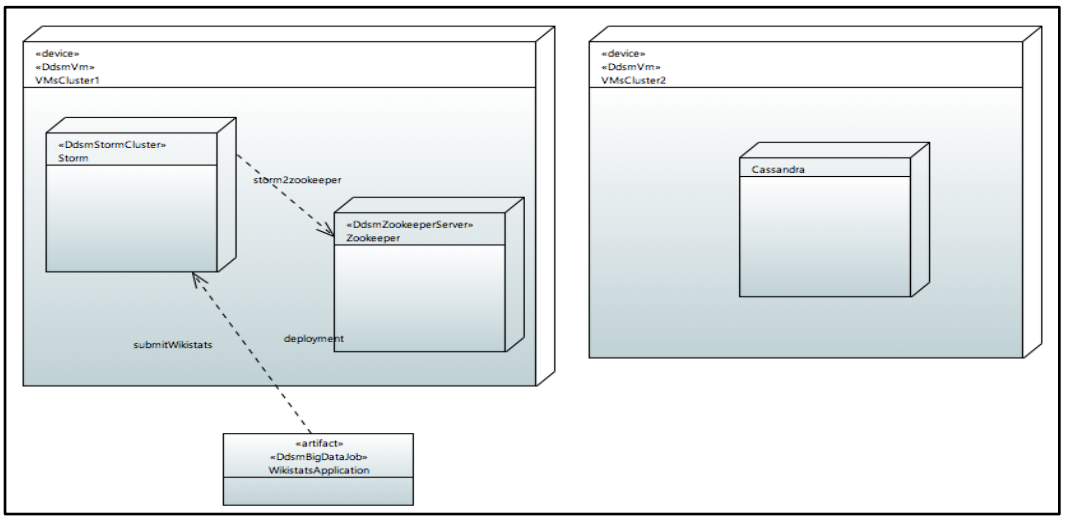

Figure 31. DICER tool: configuring the model (7).

13.5. Running the DICER Tool

Once our user has prepared the DDSM diagram following the above procedure or exploiting the DICER domain-specific language (DSL), she can use the DICER plugin to generate the associated deployable TOSCA blueprint. In order to do so our user can double click on her UML model and run through it with a DICER run configuration. In the config, the user can set the URL to contact the DICER service and specify the path to the input model and for the output model to be placed. Our user can finally click on run to generate the TOSCA blueprint and submit that to the DICE Deployment Service for further processing.

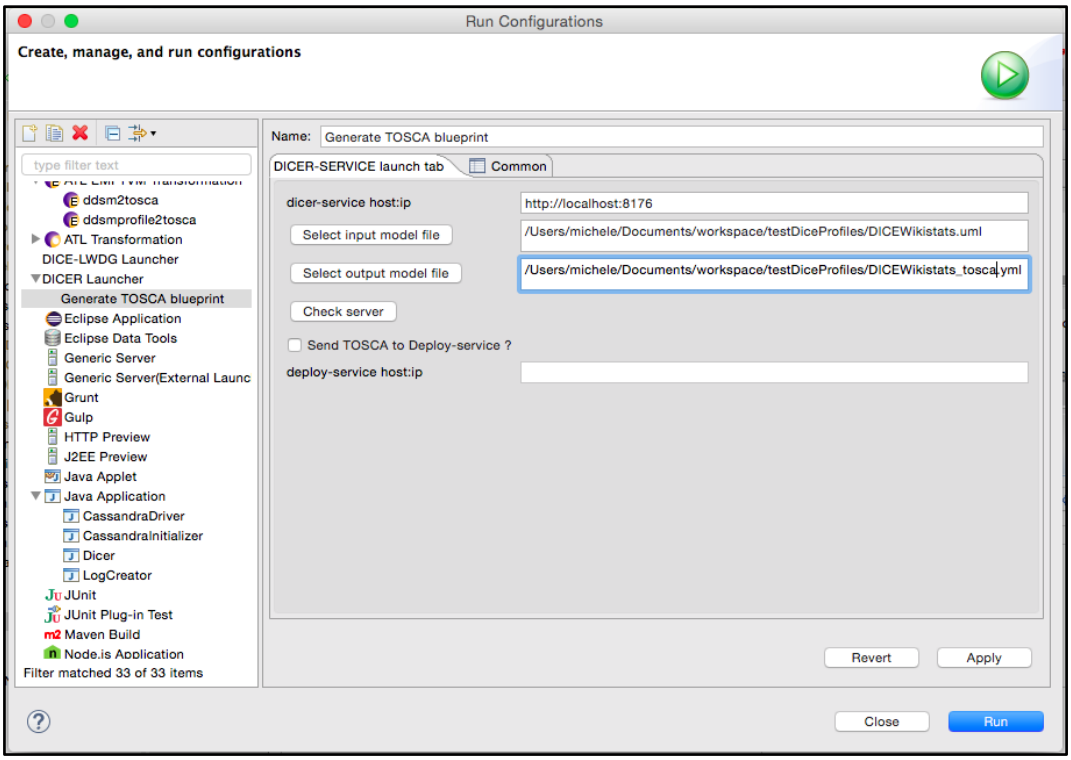

Figure 32. DICER tool: configuring the model (8).

14. Conclusions

This document described the current status of the DICE Tools.

In the next version of this deliverable (D1.6 DICE Framework – Final version), we will update the information of the DICE Tools with the progress achieved at M30 of the project.