Ref. Ares(2016)3988278 - 28/07/2016

Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Deployment abstractions

Deliverable 2.3
Version: initial

Deliverable:	D2.3
Title:	Deployment Abstractions
Editor(s):	Damian A. Tamburri (PMI), Elisabetta Di Nitto (PMI)
Contributor(s):	Michele Guerriero (PMI), Matej Artac, Tadej Borovsak (XLAB)
Reviewers:	Dana Petcu (IEAT), Craig Sheridan (FLEXI)
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-July-2016
Status:	Final version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2016, DICE consortium – All rights reserved

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

Executive summary

Data intensive applications are rapidly becoming key enablers for any industry, from automotive to entertainment to aerospace. However, requires continuous architectural design, framework/infrastructure configuration and deployment-testing to fine tune software and underlying resources by means of (re-)configuration. The goals behind the DICE project are to support the above scenarios with model-driven, DevOps-fashioned tools and methods for actionable and continuous data-intensive design and deployment.

This technical report presents a salient part of DICE, i.e., its own model-driven deployment tool called DICER, that stands for “DICE Rollout".

DICER speeds up the key phases of data intensive continuous architecting by aiding the configuration of data intensive frameworks and underlying infrastructures while fully automating the generation of deployment blueprints from design models.

DICER produces actionable blueprints in TOSCA, the “Topology and Orchestration Specification for Cloud Applications" to guarantee executability on any TOSCA-enabled orchestrator (including our own deployment service). This report also outlines the key foundations behind DICER and its technological tenets, namely, the MODACloudsML modelling notation for cloud applications as well as the TOSCA standard meta-model of our own device.

Finally, evaluating DICER with case-study research, we concluded that it does speed up considerably the work of data intensive software designers but much work still lies ahead for fully-automated continuous data intensive architecting.

Copyright © 2016, DICE consortium – All rights reserved

Glossary

DICE	Data-Intensive Cloud Applications with iterative quality enhancements
IDE	Integrated Development Environment
MDE	Model-Driven Engineering
UML	Unified Modelling Language
DICER	DICE Rollout Tool
TOSCA	Topology and Orchestration Specification for Cloud Applications
DPIM	DICE Platform Independent Model
DTSM	DICE Technology Specific Model
DDSM	DICE Deployment Specific Model

Copyright © 2016, DICE consortium – All rights reserved

Contents

Executive summary . . . 3

Glossary . . . 4

Table of Contents . . . 5

List of Figures . . . 7

List of Tables . . . 7

1 Introduction . . . 8
- 1.1 Objectives . . . 9
- 1.2 Structure of the deliverable . . . 9

2 Achievements . . . 10
- 2.1 Achievement 1 . . . 10
- 2.2 Achievement 2 . . . 10

3 DDSM and deployment modelling: State of the art overview . . . 11
- 3.1 Model-Driven Engineering for data intensive applications . . . 11
- 3.2 Deployment Modelling and Automation . . . 11

4 DICE Approach for Deployment Modeling . . . 13

5 Modelling for Deployment . . . 15

6 Generating TOSCA Blueprints with Model to Text Transformations . . . 20
- 6.1 Deploying DICER Blueprints: DICER Deployment Service . . . 22
- 6.2 Application deployments from methodological perspective . . . 23

7 DDSM and DICER in Action: Evaluation . . . 25
- 7.1 Deploying TOSCA Storm Blueprints . . . 25
- 7.2 Evaluation Objectives and Methods . . . 25
- 7.3 Evaluation by Self-Ethnography . . . 27
  - 7.3.1 Evaluating DICER with Hadoop Map Reduce Applications . . . 28
  - 7.3.2 Evaluating DICER with Oryx 2 Applications . . . 29
- 7.4 Evaluation by Case-Study Research . . . 29

8 Discussion and Observations . . . 31
- 8.1 Lessons Learned . . . 31
- 8.2 DDSM and DICER Limitations . . . 31

9 Conclusions . . . 33
- 9.1 Summary . . . 33
- 9.2 Further work . . . 33

References . . . 34

Appendix A. DDSM Metamodel . . . 36
- A.1 The `DiceDomainModel::DDSM` metamodel . . . 36

Appendix A. TOSCA Metamodel . . . 41
- A.1 The `DiceDomainModel::TOSCA` metamodel . . . 41

Copyright © 2016, DICE consortium – All rights reserved

Appendix B. Name of the achievement 2. . . . . **45**

List of Figures

1 TOSCA specification, core constructs . . . 12
2 DDSM Research Method, a general overview. . . . 13
3 An architecture for automated deployment blueprint creation for of data intensive applications. . 16
4 Exemplification of how the packaged structure is achieved in the DDSM frameworks. . . . 16
5 An excerpt of the *MODACloudsML4DICE* meta-model. . . . 18
6 An excerpt of the *TOSCA@Cloudify* meta-model. . . . 19
7 DICER framework, Architecture Overview. . . . 20
8 The MODAClouds4DICER model representing the Cloudify deployment of Apache Storm. . . . 26
9 The TOSCA model representing the deployment of Apache Storm, output of the *Deploy Transformation* . . . 26

List of Tables

1 `ddsm` data types . . . 36
2 The `ddsm` package . . . 36
3 `tosca` data types . . . 41
4 The `tosca` package . . . 41

Copyright © 2016, DICE consortium – All rights reserved

1 Introduction

Big Data technologies have rapidly achieved widespread adoption thanks to their versatility and their ability to foster innovative products. Paraphrasing from Behm et al. [1]: “Be it large IT enterprises to web companies to researchers, virtually everyone is either experiencing or anticipating the benefits from harnessing Big Data technologies". Also, according to the “Worldwide Big Data Technology and Services, 2012-2015 Forecast" by IDC [2], Big Data services are expected to grow worldwide at an annual growth rate of 40% — about seven times that of the ICT market as a whole.

However, the adoption of big data technologies is not simple to achieve. At the moment there are no simple-to-use tools that support designers and operators in configuring a data intensive application to use one or more of the available big data frameworks, and in deploying it on proper resources. The result is that such designers and operators have to go through a cumbersome trials and errors continuous architecting process where they iteratively tune the configurations of data intensive applications [3] and of their underlying infrastructure [4] until they reach satisfactory results. These exercises are, at the same time, **very expensive** — e.g., ad-hoc infrastructure needs to be rented and exploited for thousands of euros — **very complex** — e.g., they have to combine the complexity of the technological framework with concerns such as privacy, legal issues, data quality, etc. — and, finally, **very slow** — e.g., the execution of said exercises is reportedly still carried out mostly manually and by trial-and-error [5].

Of course, things become even more complicated when we need to re-design and re-configure our application to change the underlying used technology. For example, consider the scenario in which you want to change your big data streaming framework, let’s say Apache Storm[1] altogether, opting for a new one, let’s say Spark[2]. In this scenario, the application design and development constraints change considerably and would force you at least to: (a) completely redesign the data intensive application using the new concepts from the new framework, i.e., Spark; (b) install the new framework according to its deployment and configuration details; (c) configure said framework and the underlying infrastructure via an orchestration engine; (d) iteratively, test the application on the infrastructure; (e) improve incrementally both the application and the deployment configurations; (f) finally, repeat points (d) and (e) until satisfactory performance manifests. In this scenario, a number of artefacts would have to be (re-)coded and/or synched manually, for example, the code of the application or the configuration scripts needed to setup the infrastructure.

This and similar scenarios also reflect the peculiar form of data intensive applications whose non functional properties depends heavily on the many framework configuration parameters and the chosen deployment structure. For example, fine-tuning a Storm application requires experimenting on 120+ framework configuration parameters[3] - these need to be configured and experimented jointly with the deployment structure of both the Storm framework and the applications/other frameworks using it. To the best of our knowledge, no tool support exists to date that allows the manipulation of data intensive applications architecture (e.g., the set of architecture elements and frameworks part of the application), framework configuration parameters and the architecture’s deployment structure.

The DICE project argues that this and similar scenarios can be supported by combining two ingredients:

(a) Model-Driven Engineering principles and tools as they may allow designers to use models for analysing and optimizing their architectural configuration, and to abstract from the specific details of big data technologies delegating to some automation steps the generation of needed code and scripts;

(b) An integrated *infrastructure-as-a-code* [5] approach as from such kind of code — possibly automatically generated from the aforementioned models — it is possible to provide precise instructions to management tools that support the automatic deployment and configuration of applications on the specified resources.

In this deliverable we present the DICE approach and the connected results along the lines described above.

[1] `http://storm.apache.org/`
[2] `http://spark.apache.org/`
[3] `https://github.com/apache/storm/blob/v0.10.0/conf/defaults.yaml`

Copyright © 2016, DICE consortium – All rights reserved

In particular, we contribute to the state of the art in automated software engineering with: (a) a model-driven continuous architecting and deployment automation framework based on a set of modelling notations [6] needed to capture the necessary concepts, relations and constraints and a set of model-transformations; (b) a supporting tool called DICER that accelerates the continuous constraints-based (re-)deployment of data intensive applications by means of model-transformation automation; (c) a supporting library of reusable infrastructure-as-a-code components that can be exploited to deploy big data frameworks by using modern orchestration engines such as Cloudify[4] or Brooklyn[5].

Evaluating these key contributions through case-study research [7], we observed that our solution simplifies and partially automates several time-consuming activities and let us conclude that the combination of our two ingredients (model-driven engineering and infrastructure-as-a-code) play a key role in providing big data engineering with speed by automation in a DevOps fashion.

1.1 Objectives

This deliverable has the following objectives:

Objective	Description
DDSM Meta-Models	elaborate on the meta-modelling foundations with which we support automated continuous deployment of DIAs.
DICER Tool	elaborate the transformation logic behind the automation for deployment of DICE-supported DIAs.

1.2 Structure of the deliverable

The rest of this deliverable is structured as follows. First, Section 2 outlines the main DICE achievements addressed in this deliverable. Second, Section 3 outlines a state of the art overview for us to contextualise the DICE results addressed in this deliverable. Further on, Sections 4, 5 and 6 outline our research solution, namely the DDSM DICE meta-model layer, the DICER tool and the methods in which they were obtained, while Section 7 evaluates our solution by means of case-study research [7]. Further on, Section 8 concludes the deliverable by elaborating on the future work we intend on the DDSM layer and DICER tool, respectively.

Finally, the appendices provide a complete reference over the DDSM meta-modelling notations, the DICER tool, and the transformational logic behind it.

[4] http://getcloudify.org/

[5] https://brooklyn.apache.org/

Copyright © 2016, DICE consortium – All rights reserved

2 Achievements

This section briefly describes the main achievements of this deliverable.

2.1 Achievement 1

We have achieved a stable version of the DICE Deployment Modelling abstractions (DDSM) by combining an edited and updated version of the MODACloudsML language grammar (called MODAClouds4DICE) with the TOSCA standard v 1.0 grammar. The DDSM model has been tested on several technologies and it contains the necessary concepts required for DICE deployment modelling.

These abstractions allow designers to produce a deployable map for the implementable view of the big data application design realised and refined within the DTSM component. Said map essentially relies on core-constructs that are common to any cloud-based application (of which big data is a subset). Similarly to the related DTSM abstraction layer (see Deliverable D2.1), DDSM abstractions come with ad-hoc deployment configuration packages which are specific per every technology specified in the DTSM component library. Designers that are satisfied with their DTSM model may use this abstraction layer to evaluate several deployment alternatives, e.g., matching ad-hoc infrastructure needs. For example, the MapReduce framework typically consists of a single master JobTracker and one slave TaskTracker per cluster-node. Besides configuring details needed to actually deploy the MapReduce job, designers may change the default operational configurations behind the MapReduce framework. Also, the designer and infrastructure engineers may define how additional Hadoop Map Reduce components such as Yarn may actively affect the deployment.

Appendix A contains an overview of all concepts and relations captured within the DDSM-specific meta-models, namely, the MODAClouds4DICE notation as well as the TOSCA standard v1 meta-model of our own design. Both meta-models are outlined in tabular form.

2.2 Achievement 2

We have achieved an initial working implementation of (a) Model-To-Model transformations that transmute models from a DTSM specification stereotyped with DDSM constructs into a TOSCA intermediate and editable format (e.g., for experienced and ad-hoc fine-tuning) as well as (b) a Model-2-Text transformation to produce an actionable TOSCA blueprint. We named this joint set of transformations the DICER tool and evaluated them by means of case-study research.

Our evaluation shows that DICER is a successful initial implementation of MDE applied to continuous modelling and continuous deployment of Data-Intensive Applications to be supported in the DICE project.

The DICER model transformation engine covers the scenario in which the designers are satisfied with their DTSM objectives and need deployment assistance. In this scenario, DICER shall create a deployable TOSCA blueprint by matching the frameworks and technologies used in the DTSM model, with the actual deployment needs, restrictions and constraints of their runtime platforms.

Appendix B contains a general overview of the DICER tool and its internal logic using snippets of commented code.

Copyright © 2016, DICE consortium – All rights reserved

3 DDSM and deployment modelling: State of the art overview

There are several works that offer foundational approaches we considered in developing the meta-modelling and domain-specific notations required to support the DDSM meta-modelling layer and the supporting DICER tool. Said works mainly reside in model-driven engineering as well as deployment modelling & automation domains.

3.1 Model-Driven Engineering for data intensive applications

Model Driven Development (MDD) is a well known approach and has been widely exploited in many areas of software engineering. Examples are the web and mobile application development, see for instance the WebRatio[6] approach, and the development of multi-cloud applications, see for instance the MODAClouds project [8], that offers a modelling approach, called MODACloudsML, to specify the constructs and concepts needed to model and deploy cloud applications and their infrastructure needs (e.g., VMs, resources, etc.).

Also, recently in the literature a number of works have been proposed which attempt to take advantage from MDD concepts and technologies within the context of Big Data application. In [9] an interesting approach is proposed with the aim to allow MDD of Hadoop MR applications. After defining a meta-model for a Hadoop MR application, which can be used to define a model of the application, the approach offers an automatic code generation mechanism. The output is a complete code scaffold, which has to be detailed by the developer of data intensive applications with the implementation of the main application-level Hadoop MR components, the Map and Reduce functions, according to placeholders in the generated code. The main goal is to demonstrate how MDD allows to dramatically reduce the accidental complexity of developing a Hadoop MR application. Similar support is offered by Stormgen [10] which aims to provide a DSL for defining Storm-based architectures, called *topologies*.

While these approaches provide a first evidence of the utility of MDD in the context of data intensive applications, they are both focused on relying on a single underlying technology. Moreover, they focus on the development phase and do not target the deployment aspects, which would require the development and operation teams to reason on the *platform nodes* supporting the execution of technological components and on their allocation to concrete computational and storage resources.

3.2 Deployment Modelling and Automation

The infrastructure as a code approach is supported today by a number of tools that offer some scripting languages to describe the configuration of complex systems and enable the execution of the resulting configuration scripts to enable the deployment, configuration, runtime management, compliance checking and the like. Two important representatives of frameworks supporting such approach are Puppet[7] and Chef[8]. Such tools, that are often called orchestrators and configuration managers, however, solve only a part of the problem, because the actual infrastructure as a code they execute have to be provided by the users or the community.

TOSCA ("Topology and Orchestration Specification for Cloud Applications") [11] is an OASIS standandization effort that aims at providing easily deployable specifications for cloud applications in all their aspects, including, but not limited to, Network Function Virtualisation, Infrastructure Monitoring and similar. Essentially, quoting from the TOSCA specification 1.0, "TOSCA [...] uses the concept of service templates to describe cloud workloads as a topology template, [...]. TOSCA further provides a type system of node types to describe the possible building blocks for constructing a service template, as well as relationship type to describe possible kinds of relations". Figure 1, extracted from the original standard[9] outlines the essential concepts within TOSCA and their respective relation. A TOSCA specification, also called *TOSCA blueprint* is an aggregate of *topology templates*, that compose together *node templates* through *relationship templates*. A node can represent an application-level component or an

[6] `http://www.webratio.com/site/content/it/home`
[7] https://puppet.com
[8] https://www.chef.io/
[9] `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html`

Copyright © 2016, DICE consortium – All rights reserved

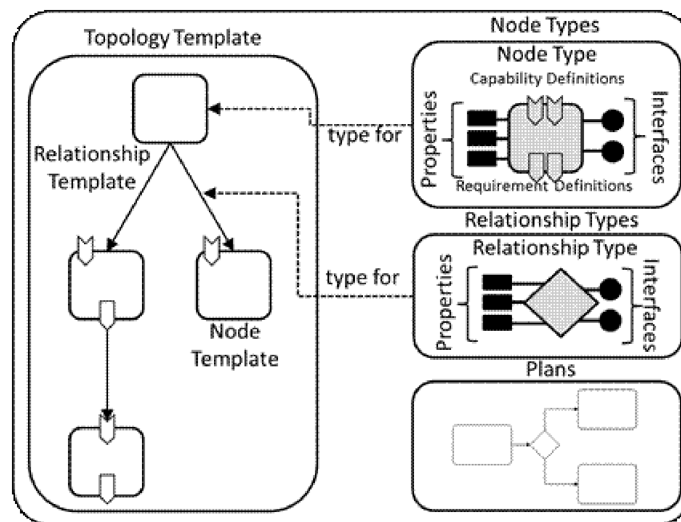

Figure 1: TOSCA specification, core constructs

infrastructural element. Its properties are defined in the *node type* specification. Relationships can have various semantics, e.g., node A *uses* node B, or node A *is running on top of* node B that are defined as part of the *relationship type* specification.

The growing interest around TOSCA is withnessed by the initiatives that are being developed around it. For instance, Cloudify[10] by GigaSpaces and Brooklyn[11] by CloudSoft are two ongoing projects that enable cloud application orchestration (deployment, configuration and execution) of topologies described in a TOSCA blueprint. Alien4Cloud[12], instead, offers simple features for modeling TOSCA topologies in a graphical way and takes care of the generation of blueprints ready to be orchestrated by Cloudify. CAMF[13] uses TOSCA to focus on three distinct management operations, particularly application description, application deployment and application monitoring[14]. Similarly, technologies such as CELAR [12] or Open-TOSCA [13], try to tackle the problem of combining model-driven solutions with TOSCA to automate deployment and speed-up the continuous architecting exercises needed for cloud applications. However, neither CELAR nor Open-TOSCA offer mature and fully usable frameworks. In general, all above frameworks have been conceived to support the definition and/or execution of blueprints of simple cloud applications. They do not offer specific support to the deployment of big data frameworks. These last ones, require specific fine-tuning of configuration parameters and constraints such as, for example, the replication factor of architectural elements.

The only approach we are aware of that offers technological parameters continuous configuration support is presented in [14]. In this paper authors show the capabilities of TOSCA to automate the deployment of scientific workflows, a specific kind of data intensive application, that can be executed in a parallel fashion according to the Map Reduce paradigm. The approach, however, is focusing exclusively on Hadoop clusters and does not consider general data intensive applications relying on different kinds of big data technologies.

In contrast to the presented approaches, we have defined in TOSCA proper node and relationship types and templates to model the specific elements associated to the most prominent big data technologies and, based on these, we offer the possibility to automatically derive TOSCA blueprints from the high level design models associated to a data intensive application. Such blueprints are then executed by our deployment service and the configuration manager that extend Cloudify and Chef tools.

[10] http://getcloudify.org/
[11] `http://brooklyn.apache.org`
[12] `http://alien4cloud.github.io/`
[13] `https://projects.eclipse.org/projects/technology.camf`
[14] `https://projects.eclipse.org/proposals/cloud-application-management-framework`

Copyright © 2016, DICE consortium – All rights reserved

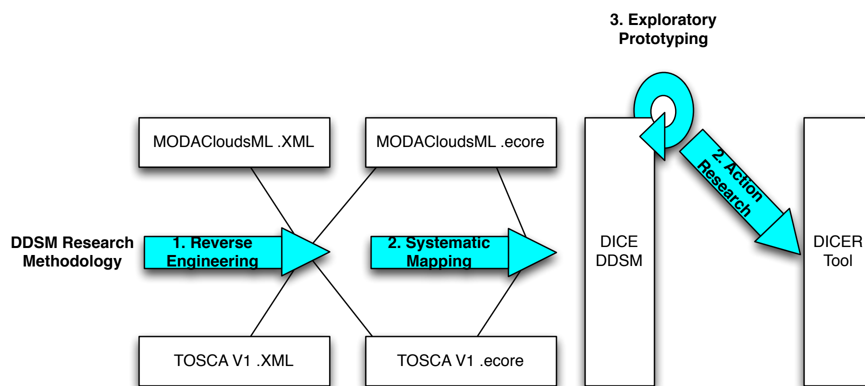

Figure 2: DDSM Research Method, a general overview.

4 DICE Approach for Deployment Modeling

From a methodological point of view, the deployment modelling and automation layer (i.e., the DDSM) and its tool support (i.e., the DICER tool) were developed harnessing a combination of (a) reverse engineering, (b) systematic mapping, (c) heavy-weight profiling and, finally, (d) explorative prototyping. Fig. 2 shows a general overview of our research methods where boxes identify either material we worked on or produced and block arrows identify research methods and approaches.

First, we applied reverse engineering principles and practices to obtain an editable meta-model for both the modelling baselines we chose to target in the scope of DDSM, namely, the MODACloudsML and the TOSCA meta-models. These baselines were selected since, on one hand, MODACloudsML represents a key result of the MODAClouds project in terms of an actionable language to fully describe and characterise quality-aware cloud applications in the scope of multi-cloud infrastructure descriptions and interoperability. On the other hand, TOSCA represents the de-facto and de-iure standard for topology and orchestration specification for cloud applications. Our research argument was that the sum of MODAcloudsML and TOSCA would yield a perfect combination of expressive power, multi-cloud tool support, interoperability and quality-awareness — these are the key principles we devised to drive the development of the DDSM layer and the prototyping of the DICER tool.

The initial working draft of said meta-models were reverse engineered from original working drafts inherited from their creators (i.e., the MODACloudsML project group and the TOSCA Technical Committee). Both meta-models were obtained in standard XMI 2.11 format and were reverse engineered using Eclipse Reflective Modelling and Model Discovery capabilities. In so doing, we obtained EMF representations of both modelling formats for the purpose of their examination and further refinement / extension within the goals and purposes of the DICE project.

Second, we conducted a systematic mapping study of concepts to be addressed within the DDSM layer and the concepts contained in the MODACloudsML and TOSCA notation meta-models. Our goal by means of this study was to determine any conceptual gaps within either notation to be addressed in the scope of DICE. As a consequence of this analysis, we applied heavy-weight profiling, that is, adding concepts to a meta-model in line with the rest of the meta-model's content, much like UML's standard heavy-weight profiling techniques [15]. In the scope of our profiling exercise, we extended both the MODACloudsML and TOSCA meta-models with concepts, notations and abstractions necessary to support Data-Intensive Applications' continuous architecting and deployment - these additional extensions were also packeted in the form of technology-specific deployment structure extensions (see bottom-left of Fig. 3).

Finally, to devise tool support and automation behind the DDSM layer towards continuous deployment we operated by means of explorative prototyping [16]. We analysed our deployment automation requirements and gave precedence to least understood and clarified requirements. Said requirements were implemented using Model2Model and Model2Text transformations featuring ATL-based DTSM models manipulation and XText auto-grammar generation (see mid-figure in Fig. 3). Following action

Copyright © 2016, DICE consortium – All rights reserved

research principles, these automations were then showcased to DICE partners in WP5 and WP6, i.e., deployment automation colleagues and case-study owners respectively.

Given that the TOSCA format is in the process of emerging, the existing vendors supporting the format cannot be fully compliant with the standard yet. Instead they do their best to cover a large part of the standard, and adapt certain aspects in their own ways. At the time of the first DICE survey [**dice-d11**], Cloudify was the strongest candidate due to its level of maturity, good community support and active development - for this very reason, we, the WP2 and WP5 team chose Cloudify as the reference technology to be further elaborated within DICE. Others include Alien4Cloud, which itself relies on Cloudify, and Apache Brooklyn, which has only recently obtained the TOSCA support.

Our aim is to have actionable blueprints, which produce real application deployments. To this end, we iterated through several phases from purely Cloudify-centric dialect of TOSCA to an increasing level of abstracting Cloudify's specifics away from the blueprints. This reverse process of implementation worked towards bringing a transformation that we could apply to other engines as well. A consequence of our action research exercise is that the notations and automation devices were refined incrementally until the preliminary maturity reported in this deliverable was achieved.

Copyright © 2016, DICE consortium – All rights reserved

5 Modelling for Deployment

Before describing into details the various components and in order to get a wider understanding of the motivation behind the DICE solution, it is worth to remind that, even if in this work we are dealing with the deployment aspects and automation of data intensive technologies and applications, in the context of Model-Driven engineering there are also two higher level modelling layers that, simply put, allow to design the architecture (i.e., the DPIM in DICE) and the technological characteristics (i.e., the DTSM in DICE) of a data-intensive application, e.g., following standard MDA approaches[15] or ad-hoc data intensive continuous design methods such as the one introduced by us in the scope of DICE and published in [17]. Figure 3 shows our high-level solution architecture in support of this reasoning.

In essence, the result of the DPIM and DTSM modelling phase is given as input to the ATL *Config Transformation* shown at the top of Fig. 3. In this work we are not interested in looking at these higher level modeling packages neither at the details of the *Config Transformation*, but it is relevant to the general understanding to briefly explain what this transformation does. Basically given a description of a data intensive application, in terms of its architecture and of the adopted technologies, the *Config Trasformation* is responsible to instantiate a first deployment solution of the application, linking all the required dependencies and deploying all the required runtime platforms. Just to make an example, if the modeled application is an Hadoop Map Reduce application, which have to retrieve the input dataset from an Apache Cassandra cluster, the *Config Transformation* will use the DDSM framework (that we are going to explain in a while) instantiating and configuring all the nodes for both Hadoop and Cassandra (i.e. the Hadoop's master node and at least a slave node) and properly linking said nodes. Moreover the transformation generates nodes for the application itself, that in the previous example will be just a single node representing the application client which submits data intensive jobs to the deployed Hadoop platform. The *Config Transformation* uses the DDSM framework according to the DICE methodology, but nothing excludes that this can be used as a component in its own right.

The DDSM framework is a modelling framework composed of the following meta-models:

First, the *MODAClouds4DICE* meta-model (top of the DDSM dotted box on the left-hand side of Fig.3) — this meta-model[16] is a transposition and an extension of the MODACloudsML meta-model adapted for the intents and purposes of data intensive deployment. MODACloudsML [18] is a language that allows to model the provisioning and deployment of multi-cloud applications exploiting a component-based approach. The main motivation behind the adoption of such a language on top of TOSCA is that we want to make the design methodology TOSCA-independent, in such a way that the designer have not to be a TOSCA-expert, nor even to be aware about TOSCA, but she should just follow the proposed methodology. Moreover the MODACloudsML language has basically the same purpose of the TOSCA standard, but it exhibits a higher level of abstraction and so results in being more user friendly. Figure 5 shows an extract of the *MODAClouds4DICE* meta-model. The main concepts are inherited directly from MODACloudsML. A MODACloudsML model is a set of *Component*s which can be owned by a Cloud provider (*ExternalComponent*s) or by the application provider (*InternalComponent*s). A *Component* can be either an application, a platform or a physical host. While an *ExternalComponent* can just provide *Port*s and *ExecutionPlatform*s, an *InternalComponent* can also require them, since it is controlled by the application provider. *Port*s and *ExecutionPlatform*s serve as a way to connect *Component*s to each other. *ProvidedPorts* and *RequiredPorts* can be linked by mean of the concept of *Relationship*, while *ProvidedExecutionPlatforms* and *RequiredExecutionPlatforms* can be linked by mean of the concept of *ExecutionBinding*. The latter could be seen as a particular type of relationship between two *Component*s which tells that one of them is executing the other. MODACloudsML has been adapted extending elements in order to capture data intensive specific concepts, e.g. systems that are usually exploited by data intensive applications such as *NoSQLStorage* solutions and *ParallelProcessingPlatform*s, which are typically composed of a *MasterNode* and one or many *SlaveNode*s.

[15] `http://www.omg.org/mda/specs.htm`

[16] A complete overview of the meta-model is discussed in the appendix and can be seen online:`PLACETHEFIGURE`

Copyright © 2016, DICE consortium – All rights reserved

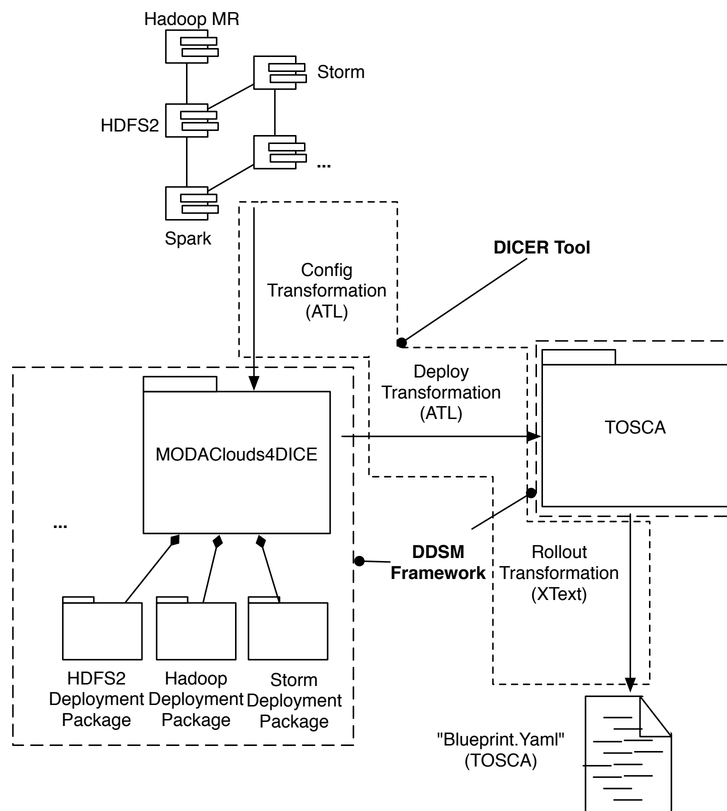

Figure 3: An architecture for automated deployment blueprint creation for of data intensive applications.

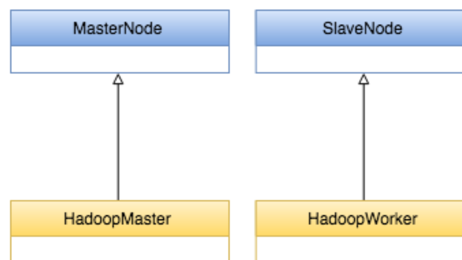

Figure 4: Exemplification of how the packaged structure is achieved in the DDSM frameworks.

Second, several data intensive deployment packages (bottom of the DDSM dotted box on the left-hand side of Fig. 3 and example extension to MODACloudsML in Fig. 4) are then used to model the deployment architectures of the data intensive technologies currently supported by our solution. Each package mainly captures the nodes, along with their *configurations*, that make up the runtime infrastructure of a given technology. For example, the Hadoop deployment package allows to instantiate an Hadoop master node and an Hadoop worker node. The packaged structure is employed in order to achieve modularity and to ease future extensions that may occur. These packages can be regarded as a further extension of the MODACloudsML language. After we identified common data intensive nodes extending the general concept of node that we have in the MODACloudsML language, each technology has its own runtime architecture composed by nodes belonging to one of the common node *types* defined in the *MODAClouds4DICE* meta-model. For example, Hadoop is composed of the *HadoopMaster* that is the *MasterNode* of a *ParallelProcessingPlatform* and one of many *HadoopWorker* that are *Slave node*s of a *ParallelProcessingPlatform* (see Fig. 5). It is worth to notice that each technological node has its own corresponding node type in the TOSCA *Technological Library* used by the Deployment Tool, so that TOSCA node templates of such a type can be declared.

Third, the *TOSCA* meta-model (See the mid-right of Fig. 3) — this meta-model captures the TOSCA

Copyright © 2016, DICE consortium – All rights reserved

grammar to enable a mapping of concepts with *MODAClouds4DICE* and the technological packages that specialise it. The *TOSCA* meta-model was obtained through reverse engineering and refined through industrial action research. Since in this work we are first of all targeting Cloudify as the supported orchestration engine, which is able to process blueprints specified not exactly using the standard TOSCA notation, but rather a dialect, we had to adapt the meta-model we derived from the standard into what we called the *TOSCA@Cloudify* meta-model[17]. Figure 6 shows and excerpt of the *TOSCA@Cloudify* meta-model, focusing of the parts that are relevant to our discussion. Here we assume the reader to be familiar at least with the basic TOSCA constructs and we don't go into details of the meta-model.

The ingredients above are vital for creating the DDSM model of any data-intensive application that aggregates the currently supported technologies. As previously mentioned, the composition of data-intensive applications is already addressed in the state of the art, e.g., in [17], and therefore is beyond the scope of this paper. Conversely, we aim at specifying the DICER tool, whose key goal is to allow model-driven continuous architecting and (re-)deployment of a data-intensive application in an IDE such as Eclipse.

[17] A complete graphical overview of the meta-model is impossible but more details on the TOSCA concepts and their relations are available in the Appendix A.

Copyright © 2016, DICE consortium – All rights reserved

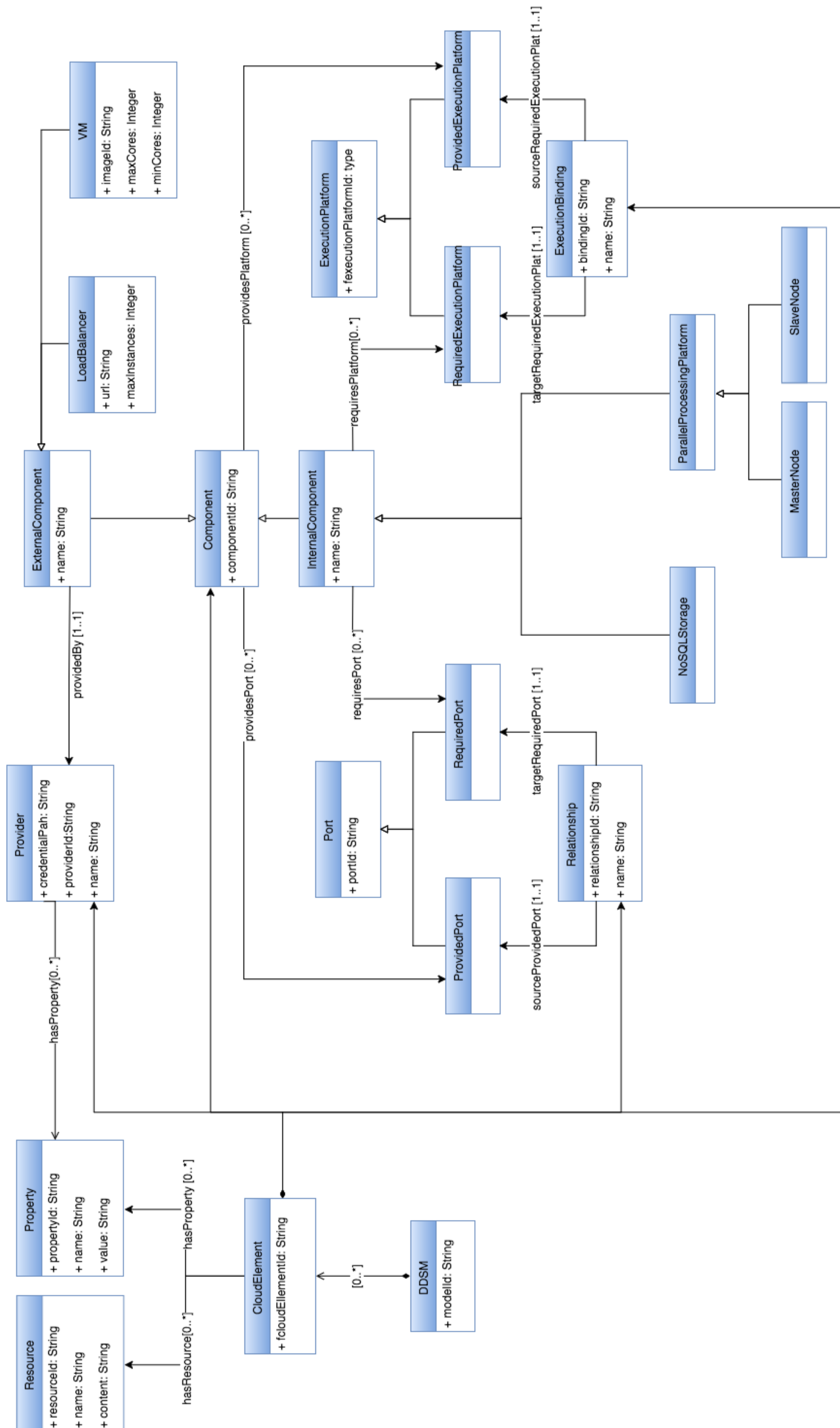

Figure 5: An excerpt of the *MODACloudsML4DICE* meta-model.

Copyright © 2016, DICE consortium – All rights reserved

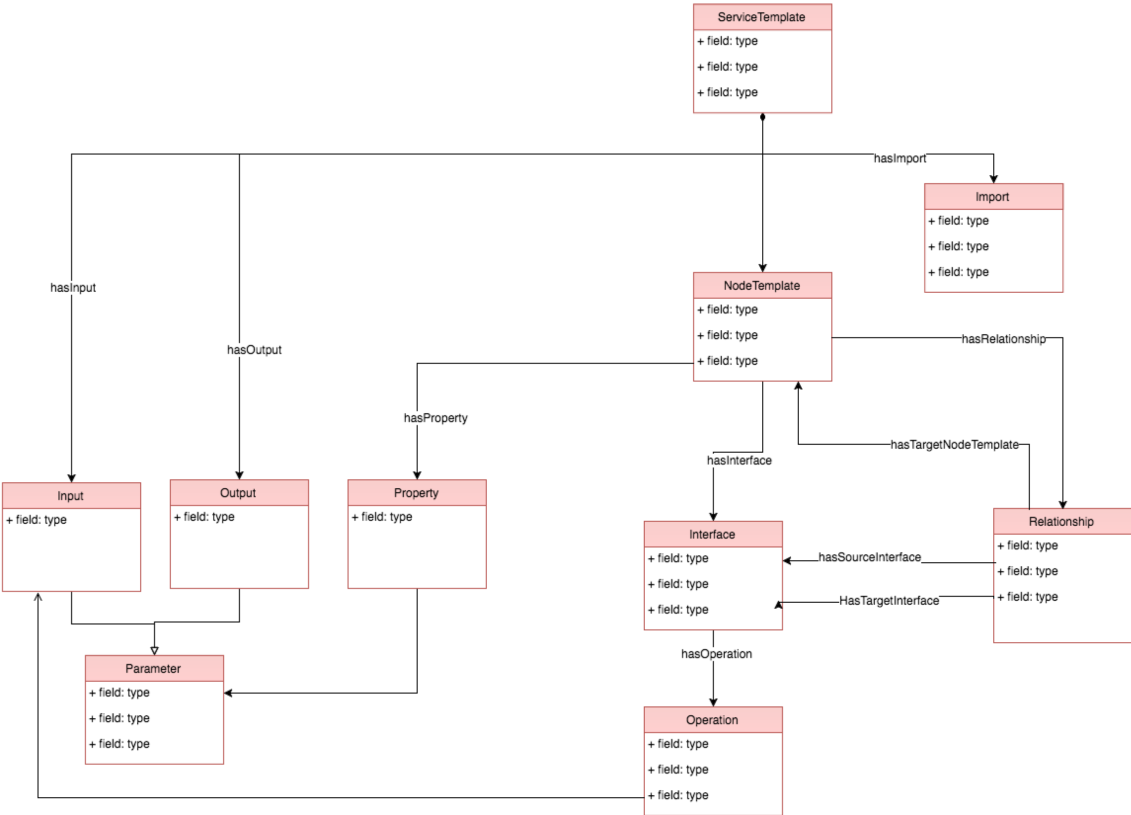

Figure 6: An excerpt of the *TOSCA@Cloudify* meta-model.

Copyright © 2016, DICE consortium – All rights reserved

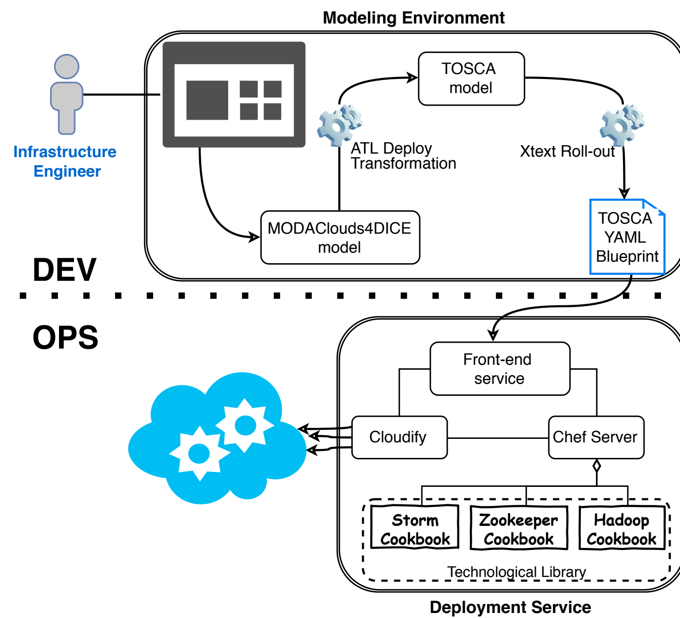

Figure 7: DICER framework, Architecture Overview.

6 Generating TOSCA Blueprints with Model to Text Transformations

From an automation perspective, DICER and the Deployment Service (i.e., matter of WP5) are essentially counterparts. On one hand, the DICE DDSM and DICER outlined in Fig. 3 offer the basis with which an actionable TOSCA blueprint is generated and fine-tuned. On the other hand, Figure 7 refines the solution architecture in Fig. 3 clarifying the relation between said solution (see Top half of Fig. 7) and the deployment service counterpart from WP5 which consumes TOSCA blueprint in its deployment and continuous integration process (see bottom-half of Fig. 7).

In summary, the overall DICE deployment architecture comprises two main components, the *Modeling Environment* (elaborated in this deliverable) and the *Deployment Service* (from WP5, Deliverable D5.1).

From a modelling perspective, through the Modelling Environment Dashboard a user (e.g., an infrastructure engineer) can drag and drop deployment elements and properly link and configure them according to the *MODAClouds4DICE* meta-model for the deployment specification of data-intensive applications. In this work, when we refer to deployment model of a data-intensive application, this includes the deployment of all the required services, technologies and of the jobs to be run over resulting platforms.

From a deployment perspective, the TOSCA model output of the Deploy Transformation is finally serialized, using an Xtext model-to-text transformation, into a deployable TOSCA YAML blueprint. This, in turn, is automatically sent to the *Deployment Service* in order to be processed for deployment, e.g., on the Flexiant test-bed. The result of this step is a working application deployed according to the topology defined in the deployment diagram.

In summary, the sole assumption behind using DICER is that data intensive designers have already prepared a component-based representation of their data intensive architecture using state-of-the-art tools and methodologies [3, 17] and featuring the MODAClouds4DICE concepts and abstractions. Conversely, the DICER Tool focuses on automatically translating said models built by the designer, into a TOSCA blueprint deployable with the Cloudify engine.

Along this path the DICER tool performs two main steps:

- **Deploy Transformation**: at this step the tool executes an ATL model-to-model transformation, which realises a translation from the *MODAClouds4DICE* language to the *TOSCA@Cloudify* language. As the source and target languages describe the same domain but adopting different notations and level of abstraction, the adopted approach in designing the *Deploy Transformation* was to look at the entities in the two meta-models describing the the same domain concept. Of

Copyright © 2016, DICE consortium – All rights reserved

course not always we found an exact one-to-one mapping between pairs of concepts. We don't want to go here into the implementation details, since this would lead into a too technical and actually not so interesting discussion, but let's just introduce an example of one main mapping that has been designed, keeping in mind the two meta-models shown in the previous Section. In the considered *TOSCA@Cloudify* meta-model we have the concepts of *NodeTemplate*, which can be used to express any kind of node, without caring about who is owning that node. Then the concept of *Relationship* can be used to link pairs of nodes and can be of different types. Thus, the *Deploy Transformation* has to treat *ExternalComponents* and *InternalComponents* of a *MODAClouds4DICE* instance model using different strategies. Among the *ExternalComponent* possible extentions, we just considered *VM*s, since the rental of computing resources is the Cloud service in which we are interested in the context of the deployment of data intensive technologies and applications. We mapped each *ExecutionBinding* into a *ContainedIn* relationship and each *Relationship* into a *ConnectedTo* relationship. Then, *ContainedIn* and *ConnectedTo* relationships are instantiated and attached only to *NodeTemplate*s generated from *InternalComponent*s, according to the *ExecutionBinding*s and *Relationship*s found in the source MODAClouds4DICE model. Listing 1 reports the ATL rule that generates a *NodeTemplate* from an *InternalComponent*. *NodeTemplate*s generated from *ExternalComponent*s have instead no attached *Relationship*s. The full ATL code is available as open source from the DICE models repository[18].

- **Rollout Transformation**: at this step the DICER tool executes an Xtext model-to-text transformation that performs a serialization of a model specified in the *TOSCA@Cloudify* language into a deployable TOSCA blueprint. The Xtext grammar has been adapted to generate a JSON document that is then converted into the YAML format.

Listing 1: An excerpt of the ATL *Deploy Tranformation*

```
lazy rule getNodeTemplatesFromInternalComp{
  from
            s : DDSM!InternalComponent
  to
            t : TOSCA!Nodetemplate(

  nodeTemplateName <- s.name,
  type <- s.oclType().toString(),
  hasRelationship <-
  s.requiredexecutionplatform ->
  collect( platform |
  thisModule.getContainedInFromExecBinding(
  DDSM!ExecutionBinding.allInstances() ->
  any(binding |
  binding.requiredexecutionplatform.platformId=
  platform.platformId
  ))).union(
  s.requiredport ->
  collect( port |
  thisModule.getConnectedToFromRelationship(
  DDSM!Relationship.allInstances() ->
  any(relationship |
  relationship.requiredport.portId=port.portId
  )))),

...
}
```

[18] `https://github.com/dice-project/DICE-Models`

Copyright © 2016, DICE consortium – All rights reserved

In our solution a user (i.e. the infrastructure engineer) can build a deployment model of her data intensive application leveraging the DDSM framework. Then, exploiting the described automation mechanisms, she can automatically get a deployable TOSCA-based blueprint, suitable for the DICE Deployment Service.

6.1 Deploying DICER Blueprints: DICER Deployment Service

The DICE Deployment Service [19] is the component, which enables the whole modeling approach in the DICE methodology to end in the live instances of technologies described in the model. Technically, it is the result of the effort in WP5, where they focus on building a technology library capable of configuring technologies supported in DICE. Methodologically, it is a natural extension of the DICER tool. This is apparent because the deployment tool is tuned to receive the kind of input that the DICER outputs.

The TOSCA blueprint produced by the DICER contains a) general declarations, which are platform-dependent, b) topology of nodes and their relationships, describing what services and applications need to be in the topology and what specific configuration parameters should apply, c) and a set of output parameters from the topology's instanciation (e.g., URLs of the applications, service access points, etc.). This effectively means that the blueprint contains only the *declaration* of the topology to be deployed. We offload all the responsibility of implementing the procedures for deploying the services to the WP5's DICE technology library.

In our proposed solution, the Deployment Service is a persistent service deployed and configured by an administrator of the development team. As the Fig. 7 shows, it consists of the following components: (1) a front-end service, (2) an orchestration engine, and (3) a technology library. The front-end service responds to client requests, abstracting specifics of the cloud orchestration engine, i.e., the Cloudify[19]. The technology library represents the main content of the Deployment Service, containing declarations of the supported technologies and the recipes for their deployment, installation and configuration. The technology library enables that DICER produces blueprints, which are largely engine agnostic. It can therefore focus on representing the service topology declaration and the services' parameters. As can be seen in listing 2, the blueprint does not have to contain a single line of code or a script reference.

The cloud orchestrator parses the blueprint and constructs a directed acyclic graph representation of the TOSCA topology in the blueprint. The relationships between nodes define the order in which the individual nodes get provisioned, deployed and configured. For instance, for deploying a web application, the orchestrator will first raise the virtual machines to host each of the services, then first install and configure a database engine before proceeding to install and configure the web server and the web application, which depends on the database to properly work.

The provisioning of the Cloud resources such as computation, networking and storage are a responsibility of specific platform's plug-ins of the orchestrator. Subject to the extent of the support, the orchestration is thus largely platform-agnostic. The installation and configuration of the services on top of the virtual or physical hosts is then the work of a configuration manager. In the proposed solution, we use Chef, which has an advantage of a powerful domain-specific language (DSL), which itself can be highly platform independent. Also, the Chef community already supplies many ready-made cookbooks for data intensive services. To properly work with the external orchestrator, however, we had to modify and extend them to separate all the stages of configuration, including: (a) installation, which places the executable files and libraries in their proper locations, but does not start the programs yet; (b) configuration, which populates the configuration files with specific values dynamically collected in the process of orchestration (e.g., a web application needs to know the address of the database engine); (c) starting the services and programs; (d) stopping the services and programs, needed as the first step of the scaling down or cleaning up actions, and removing the executable files, libraries and configuration files.

To the best of our knowledge, most of the community-provided cookbooks only provide installation and configuration steps, while the other steps (including in some cases the starting of the services) are left to the user to perform manually or using some other means.

It is worth to note that the nodes a user can instantiate while using the Modeling Environment strictly

[19]`http://getcloudify.org/`

Copyright © 2016, DICE consortium – All rights reserved

reflect the TOSCA node types defined in the Technological Library of the Deployment Service. As a result, the Modeling Environment can produce blueprints, which declare only the topology of the application and some deployment configuration parameters, while all the deployment steps implementation is hidden away.

When the orchestrator finishes with the deployment, it returns any output information specified by the application. This includes information such as the address of the web application and any other important information, which cannot be known in advance before the proper resources materialise in the deploy.

6.2 Application deployments from methodological perspective

The concept of the Deployment Service is simple: in the scope of the developers' environment it represents a service specialized in handling the parts of the applications' life-cycle which are concerned with initialization and dismantling of the deployment. Therefore it counts as a support service, similar to code repository, issue tracker or Continuous Integration services. With the support of the underlying testbed platform (OpenStack, FCO etc.), the users (e.g., developers, system architects and others) have available a small number of actions: deploy an application according to given blueprint, re-deploy the application, reconfigure the application without undeploying it, or undeploy the application. In other words, the number of manual actions involved is at the minimum. The users do *not* have to ask the administrators to manually start up virtual machines or install parts of the service in each virtual machine whenever they want to perform a deployment.

To get these benefits, there is however an inital cost of using the DICE Deployment Service, and that is the need to deploy (or, bootstrap) the service itself into the development environment. Unlike the DICER tool, which itself is a relatively simple tool to install and use from the user's depelopment machine[20], the DICE Deployment Service requires the Cloudify engine to be installed first. This, in turn, needs to be able to converse with the testbed platform's API, enabling it to handle creation or deletion of the computational, network, storage and other resources. The bootstrapping of the deplotment service therefore requires a moderate number of steps to perform, in particular the collection of the platform's parameters.

Arguably, this process could be more arduous when compared with setting up a simple Storm cluster. However, in the long term the investment pays off as soon as the users deploy or redeploy their applications more than a couple of times.

For the teams employing agile principles and following the DevOps approaches, this threshold will be quickly crossed. There is a great benefit in using the DICER and the DICE Deployment Service in the Continuous Integration workflow using Jenkins or Atlassian Jira where the tool helps perform repeatable and frequent deployments of the applications. The topology of the application could stay the same or, more likely, gradually evolve as new features are added.

The deployment tool, further, enables other DICE functionality such as the Configuration Optimization and Quality Testing. Therefore it doesn't represent the final tool in the methodology, but supports further processes, which feed the results back to the earlier stages of the DICE methodology.

[20]a hosted version with a RESTful interface is available as well

Copyright © 2016, DICE consortium – All rights reserved

Listing 2: An excerpt of the generated TOSCA blueprint for Apache Hadoop.

```
# Imports, inputs and ouputs sections omitted.
node_templates:

 # Hadoop name node setup
 namenode_firewall:
  type: dice.firewall_rules.hadoop.namenode
 namenode_host:
  type: dice.hosts.medium_host
  relationships:
   - type:    dice.relationships.protected_by
     target:  namenode_firewall
 namenode:
  type: dice.components.hadoop.namenode
  relationships:
   - type:    dice.relationships.contained_in
     target:  namenode_host

 # Hadoop resource manager
 resourcemanager_host:
  type: dice.hosts.medium_host
 resourcemanager:
  type: dice.components.hadoop.resourcemanager
  relationships:
   - type:    dice.relationships.contained_in
     target:  resourcemanager_host

 # Workers (data node + node manager combined)
 worker_host:
  type: dice.hosts.large_host
  instances:
   deploy: 3
 datanode:
  type: dice.components.hadoop.datanode
  relationships:
   - type:    dice.relationships.contained_in
     target:  worker_host
   - type:    dice.hadoop.dn_connected_to_nn
     target:  namenode
 nodemanager:
  type: dice.components.hadoop.nodemanager
  relationships:
   - type:    dice.relationships.contained_in
     target:  worker_host
   - type:    dice.hadoop.nm_connected_to_rm
     target:  resourcemanager
```

Copyright © 2016, DICE consortium – All rights reserved

7 DDSM and DICER in Action: Evaluation

In this Section we show the experiments in deploying the Apache Storm Framework[21] using the DDSM and DICER together. Later on, we show our early evaluation and assessment of the DICER tool in combination with the DCIE Deployment service for the purpose of establishing its usefulness in the scope of DICE.

7.1 Deploying TOSCA Storm Blueprints

Since Storm is one of the currently supported technologies in DICER, both the *Technology Library* of the Deployment Service and the *MODAClouds4DICE* meta-model (along with the *Deploy Transformation*) offer an ad-hoc Storm deployment package, containing Storm deployment nodes and their relation. It is then possible for a user to instantiate said node, using the DDSM framework, in her deployment model. In the case of Storm, a typical application deployment is composed of a master node called *Nimbus* and slave nodes called *Supervisor*, that have to be connected with the master node. Moreover Storm depends on Zookeeper, which has to be deployed too. A simple DICER model for this scenario is shown in Figure 8, which includes 3 *InternalComponent*s, the *StormNimbus*, one *StormSupervisor* and *Zookeeper* (single-instance), and 3 *MediumHost*s. The 3 *ExecutionBinding*s model the hosting of each *InternalComponent* on one of the available nodes, while the 3 *Relationship*s connect the 3 *InternalComponent*s. In particular *StormSupervisor* have to be connected to *StormNimbus* and both of them have to be connected to *Zookeeper*. The model in question can be further refined configuring (if needed) ad-hoc framework operations, options and defaults.

Once the user has completed this refinement, she can run the *Deploy Transformation*, obtaining the TOSCA model[22] exemplified in Figure 9 and, finally, using the *Rollout Transformation*, she can get the deployable TOSCA blueprint (in YAML format[23]), which can be sent to the DICER Deployment Service in order to start the actual deployment. An extract of the output blueprint is shown in Listing 3.

The DICER Deployment Service invokes Cloudify, which interprets the nodes and their relationships as a directed acyclic graph. As a part of the orchestration, it executes a set of pre-defined steps in the order of their dependencies. The Listing 4 shows an excerpt from the log that Cloudify emits during a typical installation run. It normally first starts the virtual machines (labelled `storm_vm`, `zookeeper_vm` and `storm_nimbus_vm`). This happens concurrently, because the nodes representing virtual machines have no interdependencies, thus the logs show them in an arbitrary order. Next, the services get installed, starting with the `zookeeper` service, proceeding with the `storm_nimbus` service and finishing with the `storm` service. The steps in the Listing 4 labeled with two node names (e.g., `storm_nimbus->storm_nimbus_vm`) indicate some relation between the depending nodes got implemented. This could be a note for Cloudify to install a service on a specific virtual machine, or a signal to Chef to configure a depending service to connect to a target service.

When the deployment tool finishes, the user receives, e.g., a functional Storm cluster ready to receive Storm topologies. The deploys also include a HDFS node on top of zookeeper and storm — entire deployment from blueprint feeding to up-time took only 20 minutes on the FCO infrastructure.

7.2 Evaluation Objectives and Methods

The objective of our evaluation efforts was twofold.

On one hand, we concentrated on understanding the length of time saved by using our research solution (i.e., the DDSM framework together with the DICER tool) with respect to deploying and incrementally refining big-data applications without such support.

On the other hand, jointly with industrial stakeholders part of DICE we wanted to understand to what degree does this version of the DDSM layer and supporting DICER tool satisfy (a) our initial DDSM

[21] `http://storm.apache.org/`

[22] A complete overview of the model is available online:`PLACEFIGUREHERE`

[23] `http://yaml.org/`

Figure 8: The MODAClouds4DICER model representing the Cloudify deployment of Apache Storm.

Figure 9: The TOSCA model representing the deployment of Apache Storm, output of the *Deploy Transformation*

Copyright © 2016, DICE consortium – All rights reserved

Listing 3: An excerpt of the generated TOSCA blueprint for Apache Storm.

```
# Imports, inputs and ouputs sections omitted.
node_templates:
 storm_nimbus:
  type: dice.storm_nimbus
  relationships:
   - type: cloudify.relationships.contained_in
     target: storm_nimbus_vm
     source_interfaces: ...
   - type: cloudify.relationships.connected_to
     target: zookeeper
     source_interfaces: ...

 zookeeper:
  type: dice.zookeeper
  relationships:
   - type: cloudify.relationships.contained_in
     target: zookeeper_vm
     source_interfaces: ...

 storm:
  type: dice.storm
  relationships:
   - type: cloudify.relationships.contained_in
     target: storm_vm
   - type: cloudify.relationships.connected_to
     target: zookeeper
     source_interfaces: ...
   - type: cloudify.relationships.connected_to
     target: storm_nimbus
     source_interfaces: ...

 storm_nimbus_vm: ...
 storm_vm: ...
 zookeeper_vm: ...
```

requirements as mapped to our industrial stakeholders' scenario and concerns and (b) any emerging deployment automation priorities and concerns from DICE case-study owners.

As a consequence, our evaluation was twofold.

First, through self-ethnography [20] we captured the tasks we carried out in deploying and improving big-data applications without any external tool and/or automation. For said tasks, we captured the average chronometric cost from two perspectives: (a) unexperienced students (two students were involved); (b) very experienced professionals and infrastructure engineers (two infrastructure engineers were involved). Subsection 7.3 outlines evaluation results from this perspective.

Second, we conducted case-study research featuring 7 interviews and 2 focus groups with partners in PRO and ATC. Our case-study exercise was aimed at establishing the validity of our research solution in the DDSM and supporting tool DICER against industrial concerns part of the DICE consortium and in line with requirements for DevOps-based industrial-strength solutions aimed at increasing market-speed and quality-aware continuous integration for the partners in question. Subsection 7.4 outlines evaluation results from this second perspective.

7.3 Evaluation by Self-Ethnography

Self-ethnography is the qualitative empirical evaluation of personal experience concerning the elaboration of a treatment [20]. In our case, self-ethnography meant to systematically capture every task

Copyright © 2016, DICE consortium – All rights reserved

Listing 4: An excerpt of Storm topology orchestration steps.

```
[zookeeper_vm] Creating node
[storm_vm] Creating node
[storm_nimbus_vm] Creating node
[storm_vm] Configuring node
[storm_vm] Starting node
[zookeeper_vm] Configuring node
[zookeeper_vm] Starting node
[storm_nimbus_vm] Configuring node
[storm_nimbus_vm] Starting node

[zookeeper] Creating node
[zookeeper->zookeeper_vm|preconfigure] Task
    succeeded
[zookeeper] Configuring node
[zookeeper] Starting node
[storm_nimbus] Creating node
[storm_nimbus->storm_nimbus_vm|preconfigure]
    Task succeeded
[storm_nimbus->zookeeper|preconfigure] Task
    succeeded
[storm_nimbus] Configuring node
[storm_nimbus] Starting node
[storm] Creating node
[storm->storm_vm|preconfigure] Task succeeded
[storm->storm_nimbus|preconfigure] Task
    succeeded
[storm->zookeeper|preconfigure] Task succeeded
[storm] Configuring node
[storm] Starting node
```

required to build a simple data intensive application and deploy it in action on a simple cluster without the intervention of any external tooling beyond tools similar to DICER. For every such task, we systematically reported the chronometric cost required to carry out the task. We operated this exercise on three technologies, namely, Storm, Hadoop Map Reduce and Oryx 2. This exercise offered a baseline to compare with DICER to find out where and how does DICER automated support actually help. In the following we report our experiments focusing on Hadoop Map Reduce and Oryx 2. Subsequently, we illustrate a Storm deploy in further detail. As a general note to allow further comparisons, it is work noting that producing a deployable blueprint using DICER without any ad-hoc configuration of the deployment structure or technological models previously elaborated is virtually instantaneous (depending on the size of architecture and resulting models). Conversely, the process of turning a component-based design into a deployable blueprint with ad-hoc configurations of technological parameters and deployment lasted up to an entire day with our industrial stakeholders.

7.3.1 Evaluating DICER with Hadoop Map Reduce Applications

Based on our experience, (re-)architecting and (re-)deploying Map Reduce applications without the intervention of DICER requires, at least: (a) creating and properly configuring virtual machines — 1 hour; (b) installing Hadoop files with requirements — 30 minutes per VM; (c) configuring machines to work in union and functioning according to Hadoop — 10 minutes per VM.

Following the above estimates, installing a small cluster of five virtual machines (name node, resource manager and three worker nodes) takes approximately 4 hours. Note that this estimate assumes that user is experienced and using some form of assisted installation procedure/script available for Hadoop files and is familiar with various settings that are needed to be set when connection all of the

Copyright © 2016, DICE consortium – All rights reserved

Hadoop components together. Conversely, the same exercise carried out by students without any prior knowledge, required up to a full day of effort (i.e., around 12 hours).

Adding DICER in the mix: Installing the same cluster, running the application and modify both after several runs, takes around 30 minutes and requires little or no user interaction at all with the intervention of the DICER tool, i.e., up to 8x times faster. The crucial parts of the produced blueprint are shown in listing 2.

7.3.2 Evaluating DICER with Oryx 2 Applications

In addition to working an Hadoop installation (see Sec. 7.3.1), a minimalistic Oryx installation depends on (a) a functioning Zookeeper service (b) a Kafka broker and (c) assumes Spark is available to be run on Hadoop cluster. All this amounts to: (a) familiarise with technological requirements of all of the above; (b) setting up three additional virtual machines; (c) placing Spark files on all worker nodes of Hadoop cluster; (d) setting up Zokeeper, Kafka and Oryx; (e) configuring service to work in a cohesive and collaborative union.

Using the assumptions and estimates from previous section, we estimated that an experienced user can prepare a small cluster of 8 virtual machines with the above contents and default configurations in approximately 6 hours. The same tasks took two days worth of effort when performed by students with no prior knowledge.

Adding DICER in the mix: Installing the same 8 virtual machines cluster with the help of the DICER tool takes approximately 35 minutes and requires no user interaction apart from the data intensive applications design and modelling efforts previously mentioned [17, 3] needed to specify an Oryx 2 application logic. Note that, from the DICER perspective, it is sufficient that any Oryx 2 application is modelled within a model-driven design tool — all DICER needs to find in the application model is the necessary technological decisions (in the form of standard UML stereotypes) and/or any ad-hoc technological configuration. Lastly, the modelling activities intended in our evaluation amount to instantiating and optionally fine-tuning a predefined series of stereotypes and constructs part of a technological meta-model already within DICER. The duration of this activity is at the discretion of the designer and may last from minutes (i.e., by directly instantiating the technological packages required without any manual configuration) or hours/days (in case more ad-hoc configurations and parameters are required).

7.4 Evaluation by Case-Study Research

To evaluate further the validity of our results, we applied case-study research by means of interviews of our industrial partners in DICE. The goal of our case-study was to understand the possible industrial impact of DICER and its automation possibilities in terms of continuous architecting. For the purpose of case-study research we produced component-based UML models as well as blueprints for two industrial case-studies featured in the DICE consortium and owned by our partners within DICE. The exercise of modelling was carried out while keeping track of the actual time to deploy, modify and re-deploy said blueprints for the purpose of continuous architecting. These models and blueprints were discussed in several open-ended interviews and focus-groups. The results of these exercises can be summarised in three key evaluation points, outlined below:

- First, DICER does in fact speed-up (re-)deployment and continuous architecting up to 10x–15x times (i.e., even faster than our original estimates — see Sec. 7.3.1), since industrial deployment times reported by interviewing our industrial partner at ATC and Prodevelop in DICE currently range from 1 up to 3 full days of deployment work, however, for scenarios where interoperation with previous deployment technologies is critical, the DICER tool slows down considerably the deployment process due to the difficult contiguity with previously existing deployment technologies. For example, Prodevelop often faces the problem of blending pre-existing infrastructure assets with their own solution - adding DICER in this mix regardless of previously existing formats and notations may compromise and make deployment efforts even longer. Further research should

Copyright © 2016, DICE consortium – All rights reserved

be invested in procuring more fine-grained identification of improvement “hotspots" on produced TOSCA blueprints — this is reportedly needed for several reasons but especially since migrating from a previous deployment solution to DICER requires comparing the benefits of DICER blueprints with previous scripts — this comparison needs to be instrumented with more ad-hoc mechanisms for easier identification of TOSCA scripts to be modified.

- Second, DICER and the rest of our deployment stack offer speed at the expense of heavy UML modelling which may be expensive in small-medium enterprises (SMEs). For example NETF declared that their tax-fraud detection application is supposed to be installed on client premises and never actually migrated. As a consequence, part of the effort of modelling using familiar yet elaborate notations for DTSM and DDSM may go wasted due to lack of further and incremental use of the models themselves. Further research should be invested in studying and refining model2model transformations that increase the automation and convenience behind using DICER, e.g., to increase interoperation between previously existing artefacts and other modelling technologies. This effort could try to address seamless and possibly effortless migration or modelling information interchange with DICER.

- Third, finally, DICER offers a very valuable component-based solution that (a) aids the composition of big-data designs in a component-based fashion (b) encourages avoiding expensive trial-and-error big-data architecting exercises (c) does in fact assist automated continuous big data architecting, however, the framework biases the adoption of continuous big data architecting towards technologies currently supported in DICER. A number of other technologies are not addressed but are currently being used or considered by our use-case owners, e.g., Drools, RabbitMQ and Flink. Further research should be invested in making the extension of DICER as effortless as possible so that the inclusion of more technologies is made easier.

Copyright © 2016, DICE consortium – All rights reserved

8 Discussion and Observations

This section outlines discussions, observations and limitations for the current Deployment Modelling and transformation technology we developed as part of DICE.

8.1 Lessons Learned

First, we learned that no currently available orchestrator engine supports the full TOSCA standard but rather continuous architecting in technologies such as Cloudify or Brooklyn is assisted by means of TOSCA “dialects". Consequently, for example, we were forced to design DICER so that it works with mainstream technology (i.e., Cloudify and Brooklyn) capturing differences and deviations where possible[24]. For this reason, the current version cannot claim 100% TOSCA standard compliance. On the other hand, we prepared a version of the DICER tool which does in fact produce 100% TOSCA-standard blueprints. However, this version of DICER is used for experimental purposes only since it does not produce “actionable" blueprints.

Second, we learned that automated solutions need to cope heavily with previously existing artefacts and solutions. In designing DICER we were questioned several times as to the degree of compatibility between DICER and its modelling framework with technologies such as Chef recipes[25] or similar previously existing deployment technology. Although we are also using Chef recipes as part of DICER, we bring our own cookbooks, the use of which is completely transparent to the DICER end-user. Including users’ cookbooks is at the time of writing not possible without changing the internal logic of DICER.

Third, since deployment tools can execute most of the operations in parallel, the time to deploy a distributed application is only dependent on the number of logical parts of the application that rely on each other. For a Hadoop cluster, for example, there are basically only two groups of tasks: installation of management nodes and installation of worker nodes. Number of nodes in each group is not important, since all of the virtual machines can be prepared in parallel. There might be some limitations present in the underlying infrastructure that will host the deployed application, but the deployment tools by itself do not impose any limitation on parallel execution of independent tasks.

Fourth, the DDSM framework and DICER tools were designed for extensibility which is one of the key advantages of the DDSM framework altogether. As previously elaborated (see Sec. 6.1), different data intensive technologies are represented by independent meta-models, so one can add as many as new technologies needed on top of the technologies already in the package library. First, following the logic outlined in [17], designers need to add a package specifying the technological implementation details needed to design application logic with the desired technology. Subsequently, designers need to extend the MODAClouds4DICE meta-model with an additional meta-model for the technology to be added. A new meta-model needs to contain: (a) compute nodes and relationships needed for the technology in question; (b) Chef recipes that correspond to the configuration of these compute nodes. For instance, in case of Hadoop MR, we extended the MODAClouds4DICE computeNode to specify a MasterNode and a SlaveNode as computational nodes. The DDSM framework[26] currently provides an extension template that can be used as a start for the above extension exercise.

8.2 DDSM and DICER Limitations

First of all, DICER was evaluated by means of qualitative evaluation alone — this type of evaluation does confirm its usefulness but cannot be generalised to any possible DICER application scenario for several reasons: (a) it involves the biases intrinsic to human intervention; (b) it does not rely on accurate machine-based time-estimations but on chronometric observations; (c) it uses interviews and opinion-based observations by industry stakeholders. More ad-hoc industrial validation, perhaps involving larger companies and larger product lines based on big data frameworks should be invested to confirm and

[24] a running-work document is being shared and maintained between WP2 and WP5 leaders and can be inspected here: `http://tinyurl.com/zgl5ff3`

[25] `https://www.chef.io/chef/`

[26] available here:`https://github.com/dice-project/DICE-Models`

Copyright © 2016, DICE consortium – All rights reserved

generalise the validity of DICER.

Second, DICER is limited by the technological stack and assumptions that entail its design. Even our own evaluation reports on its limitations in biasing big data solutions towards the frameworks and technologies currently supported. To mitigate this limitation, we structured the DDSM Framework part of DICER in a clear-cut packaged structure and prepared guidelines to extend this package library. Also, we plan to extend ourselves the library (a) with technological refinements such as TEZ or further big data technologies such as Cassandra or HBase.

Third, although DICER is set within a DevOps way of working, it is currently still limited in its ability to provide direct operations feedback models. For example, it is currently not possible for DICER to modify directly TOSCA blueprints or MODACloudsML models with feedback from operations. Nevertheless, within the DICE project, we are currently working to refine a direct DevOps continuous feedback, assisting DevOps in a more actionable way.

Copyright © 2016, DICE consortium – All rights reserved

9 Conclusions

9.1 Summary

Summarising, the main achievements of this deliverable in relation to its initial objectives:

Objective	Achievement
DDSM Meta-Models	We have achieved a stable version of the DICE Deployment Modelling abstractions (DDSM) by combining an edited and updated version of the MODACloudsML language grammar (called MODAClouds4DICE) with the TOSCA standard v 1.0 grammar. The DDSM model has been tested on several technologies and indeed it contains the necessary concepts required for DICE deployment modelling.
DICER Tool	We have achieved an initial working implementation of (a) Model-To-Model transformations that transmute models from a DTSM specification stereotyped with DDSM constructs into a TOSCA intermediate and editable format (e.g., for experienced and ad-hoc fine-tuning) as well as (b) a Model-2-Text transformation to produce an actionable TOSCA blueprint. We named this joint set of transformations the DICER tool and evaluated them by means of case-study research.

9.2 Further work

As further steps of the work we intend in DICE, we plan to further the evaluation of the DICER tool and DDSM thus tackling the discussed limitations about qualitative evaluation. Second, we plan to extend the technological support currently offered by the DDSM and DICER to cover the remaining technologies to be addressed within DICE as much as possible, with a focus on data-base technology which is ancillary to our current technological support, i.e., Hadoop Map Reduce, Oryx 2 and Storm.

Finally, we plan to integrate and re-evaluate the DICER tool as a means to support the end-to-end data-intensive design and continuous improvement workflow intended behind DICE. In so doing, we are currently working to refine DICER and the underlying DDSM notations as stand-alone solutions. This action serves two purposes: (a) DICER may be integrated within the IDE in no stretch from the rest of the DICE consortium's efforts; (b) DICER may be posted as a separate, de-facto model-driven DevOps support tool to big data design and development, e.g., as an Apache Project in its own accord. We are currently investigating the above actions and have achieved a decent understanding of the necessary effort required to move DICER forward. More in particular, we are discussing jointly with partners in WP5 (Deployment Services) and WP5 (case-study owners) to elaborate further minute planning. In concrete, however, DICER technological extension with a focus on DB will receive priority 1 while DICER integration with DICE IDE will receive priority 2.

Copyright © 2016, DICE consortium – All rights reserved

References

[1] Alexander Behm et al. "ASTERIX: towards a scalable, semistructured data platform for evolving-world models." In: *Distributed and Parallel Databases* 29.3 (2011), pp. 185–216. URL: http://dblp.uni-trier.de/db/journals/dpd/dpd29.html#BehmBCGLOVDPT11.

[2] Dan Vesset et al. "Worldwide Big Data Technology and Services 2012?2015 Forecast". In: *IDC Technical Report for EU H2020* 1 (Mar. 2012), pp. 1–34.

[3] Marcello M. Bersani et al. "Continuous Architecting of Stream-Based Systems". In: *Proceedings of the 25th IFIP / IEEE Working Conference on Software Architectures*. Ed. by Henry Muccini and Eric K. Harper. IEEE Computer Society, 2016, pp. 131–142.

[4] Lianping Chen. "Towards Architecting for Continuous Delivery." In: *WICSA*. Ed. by Len Bass, Patricia Lago, and Philippe Kruchten. IEEE Computer Society, 2015, pp. 131–134. ISBN: 978-1-4799-1922-2. URL: http://dblp.uni-trier.de/db/conf/wicsa/wicsa2015.html#Chen15.

[5] Constantine Aaron Cois, Joseph Yankel, and Anne Connell. "Modern DevOps: Optimizing software development through effective system interactions." In: *IPCC*. IEEE, 2014, pp. 1–7. URL: http://dblp.uni-trier.de/db/conf/ipcc/ipcc2014.html#CoisYC14.

[6] D. Emery and R. Hilliard. "Every Architecture Description Needs a Framework: Expressing Architecture Frameworks Using ISO/IEC 42010". In: *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009*. IEEE, Sept. 2009, pp. 31–40. ISBN: 978-1-4244-4984-2. DOI: 10.1109/WICSA.2009.5290789.

[7] R.K. Yin. *Case Study Research: design and methods*. Sage Publications Inc, 2003. URL: http://books.google.com/books?hl=no&lr=&id=45ADMg9AA7YC&oi=fnd&pg=PR9&dq=case+study&ots=E7EZBOLXGw&sig=v0CchBmSROOyyl8eoi3hB7YeMvg.

[8] Danilo Ardagna et al. "MODAClouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds". In: *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. MiSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 50–56. ISBN: 978-1-4673-1757-3. URL: http://dl.acm.org/citation.cfm?id=2664431.2664439.

[9] A. Rajbhoj, V. Kulkarni, and N. Bellarykar. "Early Experience with Model-Driven Development of MapReduce Based Big Data Application". In: *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*. Vol. 1. Dec. 2014, pp. 94–97. DOI: 10.1109/APSEC.2014.23.

[10] S. Santurkar, A. Arora, and K. Chandrasekaran. "Stormgen - A Domain specific Language to create ad-hoc Storm Topologies". In: *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. Sept. 2014, pp. 1621–1628. DOI: 10.15439/2014F278.

[11] Tobias Binz et al. "Portable Cloud Services Using TOSCA." In: *IEEE Internet Computing* 16.3 (2012), pp. 80–85. URL: http://dblp.uni-trier.de/db/journals/internet/internet16.html#BinzBLS12.

[12] Ioannis Giannakopoulos et al. "CELAR: Automated application elasticity platform." In: *BigData Conference*. Ed. by Jimmy Lin et al. IEEE, 2014, pp. 23–25. ISBN: 978-1-4799-5665-4. URL: http://dblp.uni-trier.de/db/conf/bigdataconf/bigdataconf2014.html#GiannakopoulosPMKTK14.

[13] Tobias Binz et al. "OpenTOSCA - A Runtime for TOSCA-Based Cloud Applications." In: *ICSOC*. Ed. by Samik Basu et al. Vol. 8274. Lecture Notes in Computer Science. Springer, 2013, pp. 692–695. ISBN: 978-3-642-45004-4. URL: http://dblp.uni-trier.de/db/conf/icsoc/icsoc2013.html#BinzBHKLNW13.

[14] R. Qasha, J. Cala, and P. Watson. "Towards Automated Workflow Deployment in the Cloud Using TOSCA". In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. June 2015, pp. 1037–1040. DOI: 10.1109/CLOUD.2015.146.

Copyright © 2016, DICE consortium – All rights reserved

[15] Jorge Enrique Prez-Martnez. “Heavyweight extensions to the UML metamodel to describe the C3 architectural style.” In: *ACM SIGSOFT Software Engineering Notes* 28.3 (May 9, 2008), p. 5. URL: http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft28.html#Perez-Martinez03.

[16] Clint Heyer and Margot Brereton. “Design from the everyday: continuously evolving, embedded exploratory prototypes.” In: *Conference on Designing Interactive Systems*. Ed. by Olav W. Bertelsen et al. ACM, 2010, pp. 282–291. ISBN: 978-1-4503-0103-9. URL: http://dblp.uni-trier.de/db/conf/ACMdis/ACMdis2010.html#HeyerB10.

[17] Michele Guerriero et al. “"Towards A Model-Driven Design Tool for Big Data Architectures"”. In: (2016).

[18] Nicolas Ferry et al. “CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications”. In: *Proceedings of IEEE/ACM UCC 2014*. 2014.

[19] DICE consortium. *D5.1 Delivery Tool - initial version*. Tech. rep. Imperial College London, 2016. URL: http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D5.1_DICE-delivery-tools-Initial-version.pdf.

[20] Martin Hammersley and Paul Atkinson. *Ethnography*. London: Routledge, 2003.

Copyright © 2016, DICE consortium – All rights reserved

Appendix A.1 DDSM Metamodels

A.1 The DiceDomainModel::DDSM metamodel

Table 1: ddsm data types

Name	Kind	Values or Description
VMSize	Enumeration	Small, Medium, Large

Table 2: The ddsm package

DICE ddsm Metamodel Element	Description	Attributes
CloudElement	Abstract class, inherit from MODACloudsML which capture common concepts shared by most of the classes specified in meta-model. For example a class extending CloudElement can have Properties and Resouces associated.	1. Attributes: • cloudElementId: String 2. Compositions: • resource: Resource • property: Property
Property	Represents a generic properties, specified by a pair of <id,value> and owned by a CloudElement.	1. Attributes: • value: String • propertyId: String
Resource	Represents a resource associated to an element which might be used to support the deployment and configuration of the such element. For instance a Resource may detail the deployment script of a CloudElement (e.g. an InternalComponent or an ExecutionBinding).	1. Attributes: • name: String • resourceId: String • value: String
Component inherits from: ddsm::CloudElement	Inherit from MODACloudsML, it represents a reusable type of cloud component (e.g. a virtual machine or an application).	1. Compositions: • providedport: ProvidedPort • providedexecutionplatform: ProvidedExecutionPlatform
InternalComponent inherits from: ddsm::CloudElement, ddsm::Component	Inherit from MODACloudsML, this represents a component that is managed by the application provider, or the developer (e.g. a Big Data application).	1. Compositions: • requiredport: RequiredPort • internalcomponent: InternalComponent • requiredexecutionplatform: RequiredExecutionPlatform

Copyright © 2016, DICE consortium – All rights reserved

ExecutionPlatform inherits from: ddsm::CloudElement	Inherited from MODACloudsML, it represents an generic hosting interface of a Component.	
Port inherits from: ddsm::CloudElement	Represents an interface (provided or required) of a Component. It is tipically used to link components in order to enable communication.	
RequiredPort inherits from: ddsm::CloudElement, ddsm::Port	A specific type of Port which specify that a Component requires to communicate and consume a features (e.g.access to a database) provided by another Component.	
ProvidedPort inherits from: ddsm::CloudElement, ddsm::Port	A specific type of Port which specify that a Component provides features (e.g.access to a database) which can be accessed by another Component.	
RequiredExecutionPlatform inherits from: ddsm::CloudElement, ddsm::ExecutionPlatform	A specific type of ExecutionPlatform providing hosting facilities (e.g. an execution environment, like a VM or a web server) to a Component.	
ProvidedExecutionPlatform inherits from: ddsm::CloudElement, ddsm::ExecutionPlatform	A specific type of ExecutionPlatform which requires hosting (e.g. a Big Data application requires a Big Data platform) from a Component.	
Relationship inherits from: ddsm::CloudElement	test generation	1. Attributes: • name: String • relationshipId: String 2. Associations: • providedport: ProvidedPort • requiredport: RequiredPort
ExecutionBinding inherits from: ddsm::CloudElement	Represents a binding between a RequiredExecutionPlatform and a ProvidedExecutionPlatform of two components, meaning that the first component will be hosted on the second one according to the specified binding.	1. Attributes: • name: String • bindingId: String 2. Associations: • requiredexecutionplatform: RequiredExecutionPlatform • providedexecutionplatform: ProvidedExecutionPlatform

Copyright © 2016, DICE consortium – All rights reserved

ExternalComponent inherits from: ddsm::CloudElement, ddsm::Component	Inherit from MODACloudsML, this represents a component that is managed by an external provider, for instance a AWS EC2 virtual machine.	1. Associations: • provider: Provider
Provider inherits from: ddsm::CloudElement	Represents a generic provider of Clouds services.	1. Attributes: • credentialsPath: String
VM inherits from: ddsm::CloudElement, ddsm::Component, ddsm::ExternalComponent	It is an specific ExternalComponent representing the well know concept of virtual machine. It is possible to the size of the VM in terms of RAM and CPU and Storage size ranges, the preferred operating system, the enabled ports, the desired public address and the number of executing instances, or the replication factor. It as been customized in the context of DICE to be able to specify DICE specific type of VM.	1. Attributes: • is64os: String • imageId: String • maxCores: String • maxRam: String • maxStorage: String • minCores: String • minRam: String • minStorage: String • os: String • privateKey: String • providerSpecificTypeName: String • securityGroup: String • sshKey: String • publicAddress: String • instances: String • genericSize: VMSize • location: String
DDSM	test generation	1. Attributes: • name: String • modelId: String 2. Compositions: • cloudelement: CloudElement
LifeCycle inherits from: ddsm::Resource	test generation	1. Attributes: • downloadCommand: String • installCommand: String • startCommand: String • stopCommand: String

Copyright © 2016, DICE consortium – All rights reserved

StormSupervisor inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Specilization of an InternalComponent introduced in the context of DICE which captures the deployment and configuration details of the Storm supervisor slave process.	
StormNimbus inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Specilization of an InternalComponent introduced in the context of DICE which captures the deployment and configuration details of the Storm nimbus master process.	
Zookeeper inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Specilization of an InternalComponent introduced in the context of DICE which captures the deployment and configuration details of a Zookeeper cluster.	
Kafka inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Specilization of an InternalComponent introduced in the context of DICE which captures the deployment and configuration details of a Kafka cluster.	
Cluster inherits from: ddsm::CloudElement, ddsm::Component, ddsm::ExternalComponent	Inherited from MODACloudsML, it represents a collection of virtual machines on a particular Provider. One Provider can host several Clusters..	1. Associations: • hasVm: VM
ClientNode inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	test generation	1. Attributes: • type: String • artifactUrl: String • mainClass: String
ChefResource inherits from: ddsm::Resource	Specific type of Resource introduced to support Chef as a configuration management system that could be used by supporting orchestration engines (e.g. the DICE Deployment Service based on Cloudify) to install and configure applications and platforms on VMs.	1. Attributes: • chefEndpoint: String • accessKey: String
YarnResourceManager inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Specilization of an InternalComponent introduced in the context of DICE which captures the deployment and configuration details of a the YARN ResourceManager master process.	

Copyright © 2016, DICE consortium – All rights reserved

YarnNodeManager inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Specilization of an InternalComponent introduced in the context of DICE which captures the deployment and configuration details of a the YARN NodeManager slave process.	

Copyright © 2016, DICE consortium – All rights reserved

Copyright © 2016, DICE consortium – All rights reserved

Appendix A.2 TOSCA Metamodels

A.1 The DiceDomainModel::TOSCA metamodel

Table 3: tosca data types

Name	Kind	Values or Description
CloudifyRelationshipType	Enumeration	ContainedIn, ConnectedTo
CloudifyInterfaceType	Enumeration	RelationshipLifecycle, Lifecycle
CloudifyCapabilityType	Enumeration	
CloudifyGroupType	Enumeration	

Table 4: The tosca package

DICE tosca Metamodel Element	Description	Attributes
NodeTemplate	A Node Template specifies the occurrence of a manageable software component as part of an application's topology model. A Node template is an instance of a specified Node Type and can provide customized properties, constraints or operations which override the defaults provided by its Node Type and its implementations. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • node_template_name: String • type: String • description: String 2. Compositions: • interfaces: Interface • properties: Property • attributes: Attribute • requirements: Requirement • relationships: Relationship • capabilities: Capability 3. Associations: • interfaces: Interface • properties: Property • attributes: Attribute • requirements: Requirement • relationships: Relationship • capabilities: Capability

Interface	An interface defines a named interface that can be associated with a Node or Relationship Type. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `name`: `String` 2. Compositions: • `operations`: `Operation` • `inputs`: `Input`
Relationship	A Relationship Template specifies the occurrence of a manageable relationship between node templates as part of an application's topology model. A Relationship template is an instance of a specified Relationship Type . For the accurate description refer to the TOSCA standard document [**toscayaml**].	1. Attributes: • `type`: `CloudifyRelationshipType` 2. Associations: • `interfaces`: `Interface` • `properties`: `Property` • `attributes`: `Attribute`
Requirement	A Requirement describes a dependency of a TOSCA Node Type or Node template which needs to be fulfilled by a matching Capability declared by another TOSCA modelable entity. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `name`: `String` • `node`: `String` • `capabiity`: `CloudifyCapabilityType`
Operation	An operation defines a named function or procedure that can be bound to an implementation artifact (e.g., a script). For the accurate description refer to the TOSCA standard document [**toscayaml**].	1. Attributes: • `operation_name`: `String` • `description`: `String` • `script`: `String` • `executor`: `String` 2. Compositions: • `operation_hasInput`: `Input`

Copyright © 2016, DICE consortium – All rights reserved

`TopologyTemplate`	A Topology Template defines the topology of a cloud application. The main ingredients of the topology template are node templates representing components of the application and relationship templates representing links between the components. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `tosca_definitions_version`: `String` 2. Compositions: • `imports`: `Import` • `outputs`: `Output` • `inputs`: `Input` • `nodeTemplates`: `NodeTemplate` • `realtionships`: `Relationship` • `groups`: `Group` • `policies`: `Policy` 3. Associations: • `imports`: `Import` • `outputs`: `Output` • `inputs`: `Input` • `nodeTemplates`: `NodeTemplate` • `realtionships`: `Relationship` • `groups`: `Group` • `policies`: `Policy`
`Import`	An import is used to locate and uniquely name another TOSCA file which has type and template definitions to be imported (included) and referenced. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `import_name`: `String` • `file`: `String` • `repository`: `String` • `namespace_uri`: `String` • `namespace_prefix`: `String`
`Group`	A group definition defines a logical grouping of node templates, typically for management purposes. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `name`: `String` • `type`: `CloudifyGroupType` • `description`: `String` • `targets`: `String` 2. Associations: • `properties`: `Property` • `interfaces`: `Interface`
`Policy`	A policy definition defines a policy that can be associated with a TOSCA topology. For the accurate description refer to the TOSCA standard document [**toscayaml**]	

Copyright © 2016, DICE consortium – All rights reserved

`Capability`	A Capability defines a named, typed set of data that can be associated with Node Type or Node Template to describe a transparent capability or feature of the software component the node describes. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `type`: `CloudifyCapabilityType` • `description`: `String` 2. Associations: • `properties`: `Property` • `attributes`: `Attribute`

Copyright © 2016, DICE consortium – All rights reserved

Appendix B. DICER Tool Snippets.

Find below a sample set of snippets extracted from the DICER source code. For the rest of the code please refer to the DICE online GitHub repository.

```
-- Sample DDSM to TOSCA M2M Transformation

-- @path oryxTOSCA=/DiceModelingProject/model/TOSCA/_12.ecore
-- @path OryxDDSM=/DiceModelingProject/model/DDSM-oryx/Cloud.ecore

module OryxDDSM2TOSCA;
create OUT : oryxTOSCA from IN : OryxDDSM;



helper context OryxDDSM!HMR_DDSM def: referenceRelationships(): Sequence
    ↪ (OryxDDSM!NodeRelationsipComponent) = self.hasRelationships.
    ↪ asSequence();
rule OryxDDSM2TOSCA{
        from
                s: OryxDDSM!Oryx2
        to
                t: oryxTOSCA!TServiceTemplate(
                        id <- s.id,
                        name <- s.oryxType,
                        hasImports <- thisModule.getimports(s),
                        topologyTemplate <- thisModule.
                            ↪ getTopologyTemplate(s)
                        --hasRelationship <- s.referenceRelationships()
                            ↪ -> collect(relashtion | thisModule.
                            ↪ getserviceRelationShips(relashtion)) --
                            ↪ this part is commented as Xlab requested
                            ↪ to move the relationships to Library

                )
}

lazy rule gethadoopCluster{
        from
                i: OryxDDSM!NodeRelationsipComponent
        to
                t: oryxTOSCA!TRelationshipType(

                        name <- i.relationName
                )
}




lazy rule getserviceRelationShips{
        from
                i: OryxDDSM!NodeRelationsipComponent
        to
                t: oryxTOSCA!TRelationshipType(

                        name <- i.relationName

                )
}
```

Copyright © 2016, DICE consortium – All rights reserved

```
helper context OryxDDSM!Oryx2 def: reference(): Sequence (OryxDDSM!input
    ↪ ) = self.hasinputs.asSequence();
helper context OryxDDSM!HMR_DDSM def: referenceMaster(): OryxDDSM!
    ↪ MasterNode = self.hasMasterNode;
helper context OryxDDSM!Oryx2 def: referenceSlaves(): OryxDDSM!SlaveNode
    ↪  = self.usesHadoopCluster.hasSlaves;
helper context OryxDDSM!Oryx2 def: referenceoutputs(): Sequence (
    ↪ OryxDDSM!output) = self.hasoutputs.asSequence();
lazy rule getTopologyTemplate{
        from
                i: OryxDDSM!Oryx2
        to
                t: oryxTOSCA!TTopologyTemplate(

                        hasInputs <- i.reference() ->  collect(inp |
                            ↪ thisModule.getinputs(inp)),
                nodeTemplate <- thisModule.getMasterNodeTemplate(i.
                    ↪ usesHadoopCluster.hasMasterNode),
                        nodeTemplate <- thisModule.
                            ↪ getMasterNodeConfigNode(i.
                            ↪ usesHadoopCluster.hasMasterNode.
                            ↪ contains_config),
                        nodeTemplate <- i.referenceSlaves()-> collect(
                            ↪ slv | thisModule.getSlaveNodeTemplate(slv)
                            ↪ ),
                        nodeTemplate <- thisModule.getMasterNodechef(i.
                            ↪ usesHadoopCluster.hasMasterNode.
                            ↪ contains_master_chef),
                        nodeTemplate <- thisModule.getMasterNodechef(i.
                            ↪ usesHadoopCluster.hasSlaves.first().
                            ↪ contains_worker_chef),
                        nodeTemplate <- thisModule.getOryxRunner(i.
                            ↪ usesHadoopCluster.hasMasterNode.
                            ↪ hasOryxRunner),
                        hasoutPuts <- i.referenceoutputs() -> collect(
                            ↪ out | thisModule.getoutputs(out))

                )
}


helper context OryxDDSM!OryxRunner def: referenceNodeHasRelations():
    ↪ OryxDDSM!relation = self.relationships;
lazy rule getOryxRunner{
        from
                i : OryxDDSM!OryxRunner
        to
                t: oryxTOSCA!TNodeTemplate(

                id <- i.name,

                --nodeHasInterface <- thisModule.
                    ↪ getMasterNodeConfigInterface(i),-- interfaces are
                    ↪ commented by Xlab request
                type <- i.type,
                nodeHasRelations <-i.referenceNodeHasRelations() ->
                    ↪ collect(rel| thisModule.
```

Copyright © 2016, DICE consortium – All rights reserved

```
                ↪ getMasterNodeConfigRelationShips(rel))

                )
}

helper context OryxDDSM!worker_chef def: referenceNodeHasRelations():
    ↪ OryxDDSM!relation = self.relationships;
helper context OryxDDSM!master_chef def: referenceNodeHasRelations():
    ↪ OryxDDSM!relation = self.relationships;
lazy rule getMasterNodechef{
        from
                i : OryxDDSM!master_chef
        to
                t: oryxTOSCA!TNodeTemplate(

                id <- i.name ,

                --nodeHasInterface <- thisModule.
                    ↪ getMasterNodeConfigInterface(i),
                type <- i.type,
                nodeHasRelations <-i.referenceNodeHasRelations() ->
                    ↪ collect(rel| thisModule.
                    ↪ getMasterNodeConfigRelationShips(rel)),
                properties  <- thisModule.getChefNodePropertiesType(i)


                )
}

lazy rule getChefNodePropertiesType{
        from
                i : OryxDDSM!master_chef
        to
                t: oryxTOSCA!PropertiesType(

                property <-  thisModule.getchefNodePropertyversion(i),
                property <-  thisModule.
                    ↪ getchefNodePropertychef_server_url(i),
                property <-  thisModule.getchefNodePropertyenvironment(i
                    ↪ ),
                property <-  thisModule.
                    ↪ getchefNodePropertyvalidation_client_name(i),

                property <-  thisModule.
                    ↪ getchefNodePropertyvalidation_key(i),
                property <-  thisModule.
                    ↪ getchefNodePropertynode_name_prefix(i),
                property <-  thisModule.
                    ↪ getchefNodePropertynode_name_suffix(i),
                property <-  thisModule.getchefNodePropertyattributes(i)
                    ↪ ,

                property <-  thisModule.getchefNodePropertyrunlists(i),
                property <-  thisModule.getchefNodePropertychefCookBooks
                    ↪ (i)

                )
}
```

Copyright © 2016, DICE consortium – All rights reserved