Ref. Ares(2016)530056 - 01/02/2016

Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Design and quality abstractions - Initial version

Deliverable 2.1

Deliverable:	D2.1
Title:	Design and quality abstractions - Initial version
Editor(s):	Abel Gómez (ZAR) and José Merseguer (ZAR)
Contributor(s):	Abel Gómez (ZAR), Michele Guerriero (PMI), José Merseguer (ZAR), Elisabetta di Nitto (PMI) and Damian A. Tamburri (PMI)
Reviewers:	Matej Artac (XLAB) and Tatiana Ustinova (IMP)
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-January-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

This document presents the initial version of the DICE design and quality abstractions. Therefore, it provides the initial versions of the DICE Models and the DICE Profile, both artifacts are briefly introduced with an executive summary and extensively described in the Appendices A and B of this document, respectively. This deliverable will be a baseline for deliverable D2.2 (*Design and quality abstractions - Final version*) due in M24. The work presented in this deliverable has been carried out within tasks T2.1 (Data-aware functional models) and T2.2 (Data-aware quality annotations).

All the artifacts described in this document are publicly available in the so-called DICE-Models Repository [**dice:models:repo**] and DICE-Profiles Repository [**dice:profile:repo**].

Glossary

DAM	Dependability Analysis and Modeling
DDSM	DICE Deployment Specific Model
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DPIM	DICE Platform Independent Model
DTSM	DICE Technology Specific Model
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MTM	Model To Model
NFP	Non-Functional Properties
UML	Unified Modelling Language
VSL	Value Specification Modeling

Contents

List of Figures

List of Tables

1 Introduction and Context

The focus of the DICE project is to define a quality-driven framework for developing data-intensive applications that leverage Big Data technologies hosted in private or public clouds. DICE offers a novel profile and tools for data-aware quality-driven development. This document describes the initial version of the DICE Profile and the DICE Metamodels, the latter are needed to develop the DICE-Profile according to the Approach described in Section 3 of this document. The DICE Metamodels and the DICE Profile, developed in the scope of WP2 as Tasks 2.1 (Data-aware functional models) and 2.2 (Data-aware quality annotations), are published in the DICE-Profiles [**dice:profile:repo**] and DICE-Models [**dice:models:repo**] repositories.

1.1 Objectives of WP2

The goal of WP2 is to provide and evaluate the necessary design abstractions for specifying data-intensive cloud applications. The work captured in this deliverable, focuses on Tasks 2.1 and 2.2 (see below), namely, the conceptual definition of the DICE design abstractions. Remaining tasks cover the specification of quality annotations, data protection and privacy constraints and will introduce or refine the DICE Platform Independent Model (DPIM), DICE Technology Specific Model (DTSM), and DICE Deployment Specific Model (DDSM) logical layers of the DICE Profile. Finally, the WP needs to define the DICE MDD methodology, which envisions the identification of the design and tool-chain usage workflow to support the continuous development of data-intensive applications.

1.2 Objectives of Task 2.1 and Task 2.2

Task 2.1 will provide DPIM and DTSM abstractions of the DICE Profile to describe data properties, data usage requirements and data transformations among other data-related concerns. Outputs of this task will be technology-specific models that include the necessary abstractions for a data-intensive application developer to specify operations performed on data, qualifying inputs and outputs.

Task 2.2 will define DPIM and DTSM abstractions (UML-based languages and profiles) to specify reliability, efficiency, and safety requirements for data-intensive applications, application subcomponents, and data usage. This task will use as baselines DAM and MARTE profiles, these were described in the DICE *State of the Art Analysis* deliverable [**dice:d1.1**].

1.3 Objectives of this document

This document presents the initial version of the DICE Metamodels and DICE Profile at DPIM and DTSM levels. DICE Metamodels are the output of Task 2.1, while the DICE Profile is the output of Task 2.2. This document serves as a baseline for deliverable D2.2 (*Design and quality abstractions - Final version*).

1.4 Structure of the document

The structure of this deliverable is as follows:

- Section 2 summarizes the requirements that Tasks 2.1 and 2.2 aim to cover.
- Section 3 presents the Approach developed for constructing the Metamodels and for developing the Profile.
- Section 4 summarizes the contribution of this deliverable at DPIM level.
- Section 5 summarizes the contribution of this deliverable at DTSM level.
- Section 6 summarizes the goals achieved, and outlines the future work.
- Appendix A details the current version of the DICE Metamodels at DPIM and DTSM levels.

- Appendix B details the current version of the DICE Profile at DPIM and DTSM levels.

2 Requirements

Deliverable D1.2 [**dice:d1.2**, **dice:d1.2:companion**], released on month 6, presented the requirements analysis for the DICE project. The outcome of the analysis was a consolidated list of requirements and the list of use cases that define the project's goals that guide the DICE technical activities.

This section recapitulates these requirements for Tasks T2.1 and T2.2.

2.1 Requirements

ID	R2.1
Title	DICE Methodological Paradigm
Priority	Must have
Description	The DICE profile and methodology shall support the incremental specification of Data-Intensive Applications (DIAs) following a Model-Driven Engineering approach, as defined in standard OMG guidelines.

ID	R2.2
Title	Abstraction Layer Origin
Priority	Must have
Description	Every abstraction layer (namely, DPIM, DTSM and DDSM) of the DICE profile MUST stem from UML.

ID	R2.3
Title	Relation with MARTE UML Profile
Priority	Must have
Description	The DICE Profile MUST define required and provided properties of a DIA as well as metrics (estimated, measured, calculated and requirements) to monitor them. Said metrics will be specifed following the MARTE NFP framework.

ID	R2.4
Title	DICE Constraints Specification
Priority	Must have
Description	The DICE Profile MUST allow definition of values of constraints (e.g., maximum cost for the DIA), properties (e.g., outgoing flow from a Storage Node) and stereotype attributes (batch and speed DIA elements) using the MARTE VSL standard.

ID	R2.5
Title	DICE Profile Performance Annotations
Priority	Must have
Description	The DICE Profile shall define annotations for performance based on the MARTE::GQAM framework.

ID	R2.6
Title	DICE Profile Reliability Annotations
Priority	Must have
Description	The DICE Profile shall define annotations for reliability based on the DAM profile.

ID	R2.7
Title	DICE Extension-Points
Priority	Must have

Description	The DTSM MUST include extension facilities. These facilities shall be used to "augment" the DICE profile with technologies beyond the DICE project assumptions (e.g., Storm, Spark, Hadoop/MR, etc.). Similarly, every technological space embedded within the DICE profile shall exist in the form of such extensions, e.g., as conceptual packages (at the DTSM layer) and refined implementation-specific packages (at the DDSM layer).

ID	R2.8
Title	DICE Profile Main DIA Concerns - Flow and Behavior
Priority	Must have
Description	The DICE Profile shall define annotations that address behavioral and flow concerns behind DIAs. Also, the DICE Profile shall define annotations for flow-control across DIAs.

ID	R2.9
Title	DICE Topologies
Priority	Must have
Description	The DTSM layer MUST support the definition of Technology-specific DIA Topologies (e.g., Namenode-Datanode-SecondaryNamenode vs. Master-Region-Zookeeper, etc.).

ID	R2.10
Title	DICE Profile Tech-Specific Constraints
Priority	Must have
Description	The DICE Profile MUST define structural and behavioral constraints typical in targeted technologies (e.g., Hadoop, Storm, Spark, etc.).

ID	R2.11
Title	DICE Profile Separation-of-Concerns
Priority	Must have
Description	The DICE Profile MUST use packages to separately tackle the description of targeted technologies in the respective profile abstraction layers (e.g., DTSM and DDSM). Said packages shall be maintained consistently

ID	R2.13
Title	DICE Profile Data Structure
Priority	Must have
Description	The DICE Profile shall define QoS annotations for data structure and its specification.

ID	R2.14
Title	DICE Profile Data Communication
Priority	Must have
Description	The DICE Profile shall define annotations to elaborate on structural and behavioral details concerning the channeling and marshalling of information across specified DIAs.

ID	R2.15
Title	DICE Profile Sub-Structures
Priority	Must have
Description	The DICE Profile shall provide annotations for specifying node nesting and replication across the structure of DIAs.

3 Research and Development Approach

As recalled in the DICE *State of the Art Analysis* deliverable [**dice:d1.1**], MDE techniques [**MDE**] and MDA in particular [**MDA**] define the typical abstraction layers for the purpose of engineering software systems using a model-centric perspective. The fundamental axiom behind this engineering paradigm is that any engineering endeavor shall be guided by at least three compounding and interoperating perspectives, namely: (a) Computational-Independent perspective; (b) a Platform-Independent perspective; (c) a Platform-Specific perspective. Using these three perspectives, one or more models can be specified to properly and systematically specify a system-to-be. In DICE these perspectives take the form of DPIM, DTSM and DDSM, respectively, while the specification language adopted is UML.

UML [**UML**] is a General Purpose Modelling Language. Therefore, it can be used to model a wide range of systems but not all of its modelling capabilities are necessarily useful in all domains or applications. Conversely, Domain-Specific Modelling Languages (DSML) are conceived for addressing the needs of specific application domains. In this regard, UML offers a solution, the so-called UML profiling mechanism [**UML**]. Profiling opens the possibility of creating DSMLs by extending or restricting UML. A Profile is then an adaptation of UML to fit a specific domain. In short, a UML Profile is made of a set of *stereotypes*, a set of *tags* and a set of related *constraints*. A stereotype is just a name that will be attached to certain elements of a UML diagram. Stereotypes have tags, we can see them as the attributes added by the stereotype.

UML has been extended with two Profiles of interest for DICE, namely MARTE [**MARTE**] and DAM [**DAM**]. MARTE (Modelling and Analysis of Real-Time and Embedded systems) provides support for the specification, design, quantitative evaluation, and verification & validation of software systems. DAM (Dependability Analysis and Modelling Profile) provides support for the dependability modelling and analysis of software systems. However, neither MARTE nor DAM has a direct support for expressing data location, data properties such as volume or transfer rates or operations that move data. Hence, addressing such lack is the main objective of the DICE Profile.

For constructing a technically correct high-quality UML profile, that covers the necessary concepts according to the DIA technologies, several steps need to be followed. First, metamodels for each abstraction level, i.e. DPIM, DTSM and DDSM, that define the concepts are needed. We have carried out this step by carefully reviewing the abstract concepts for modelling DIA, then obtaining the abstractions for the DPIM level, which conform the DICE Metamodel at DPIM level. Later, we have reviewed the different Big Data technologies addressed by DICE (e.g., Hadoop, Spark or Storm) and we have defined the abstractions of interest, consequently obtaining the DICE Metamodels at DTSM level. The last level, DDSM, will be subject of another deliverable D2.3 (*Deployment abstractions - Initial version*), due in M18.

As a second step, the DICE Profile, at DPIM and DTSM levels, was defined by mapping the concepts from the DICE domain models or DICE Metamodels to UML, MARTE and DAM. Using the DICE domain models we designed: (a) the DICE extensions (stereotypes and tags), and (b) the DICE library containing DIA specific types. We followed an iterative process for the profile definition, in which each domain class was examined, together with its attributes, associations and constraints, to identify the most suitable UML base concepts for it, as suggested in [**S07**].

While constructing the DICE Profile, following WP2 requirements, we introduced a set of stereotypes small yet expressive, that can be easily used by the software engineer. We then used guidelines from [**LETG07**] to select the subset of the domain classes that eventually were mapped to stereotypes. Moreover, several patterns proposed in [**LETG07**] were applied (e.g., the reference association pattern), that enable the creation of a profile from the domain model while keeping it consistent with the UML meta-model. The DICE stereotypes are then assets for annotating UML DIA models at the different abstraction levels. This set of stereotypes here proposed is the one for obtaining formal models and TOSCA models by M2M transformations. Currently the stereotypes proposed are useful for obtaining performance models. Later, we will extend this proposal with stereotypes useful for obtaining reliability models (Task 2.2) and TOSCA models (Task 2.3).

In the following we summarize main concerns regarding the DICE Metamodel and DICE Profile.

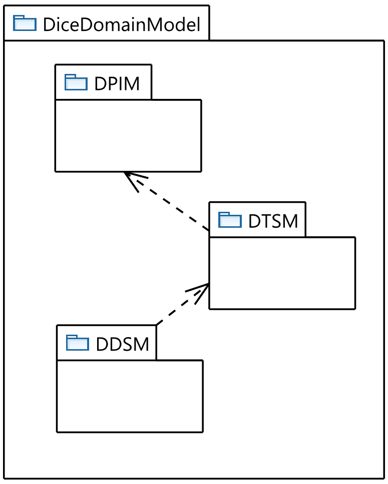

Figure 1: DICE Metamodel - High-level View

3.1 DICE Metamodel

The DICE metamodel, detailed in Appendix A of this document, is sketched in Figure 1. DICE considers one metamodel per abstraction level: DPIM, DTSM and DDSM. The rationale behind this organization is that each abstraction level keeps separated, self-contained and mutually incremental. At DPIM level the metamodel provides those abstract concepts needed for DIA modeling. At DTSM level, see Figure 2 the metamodel provides a core-DTSM metamodel and one DTSM metamodel for each technology addressed.

The logical division of the most complex abstraction layers in the DICE Profile, namely, the DICE DTSM (addressed in this deliverable) and DDSM (addressed in D2.3 deliverable) layers was arranged using a standard package-like notation. Following a systematic approach tailored from Formal Concept Analysis (FCA) [**fca**], we elicited the core-constructs common to all technologies addressed by DICE and captured said constructs in a core package, to be used by all technological extensions. In addition, specific technological extensions, e.g., Hadoop MR or Storm, were self-contained into separate pack-

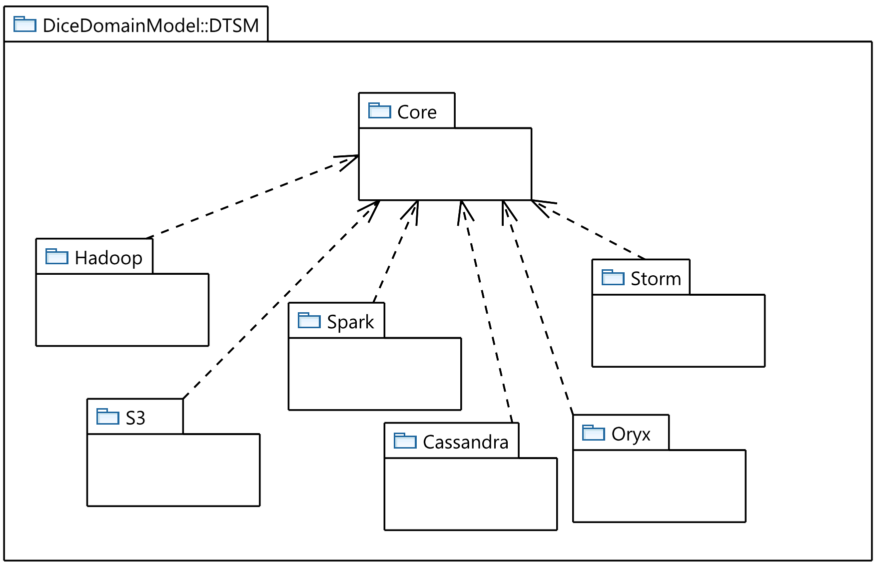

Figure 2: DICE Metamodel - DTSM View

ages, thus allowing to keep them transparent to less expert users. Nevertheless, DICE will strive to make available these packages as possibly instantiable and modifiable constructs, e.g., to accommodate the needs of more experienced users.

3.2 DICE Profile

For each aforementioned metamodel we propose a mapping for obtaining the DICE Profile, as detailed in Appendix B of this document. The DICE Metamodel is a live artifact that will evolve during the project, hence the mapping here presented will necessarily evolve with the DICE Metamodel. Figure 3 offers a high-level view of the DICE Profile, which basically contains the DICE Library and the DICE Extensions.

The DICE Library, detailed in Figure 4, contains basic and complex DIA types. We have imported the DAM library, which also imports the basic Non-Functional Properties (NFP) types from the MARTE library, for the definition of these types. In particular, the MARTE NFPs sub-profile is applied to the definition of new basic DIA types and the Value Specification Modeling (VSL) sub-profile to the definition of the complex ones.

The DICE Extensions package, detailed in Figure 5, provides the domain expert with a set of stereotypes to be applied at model specification level, i.e., the stereotypes necessary to represent the different system views in concrete UML models.

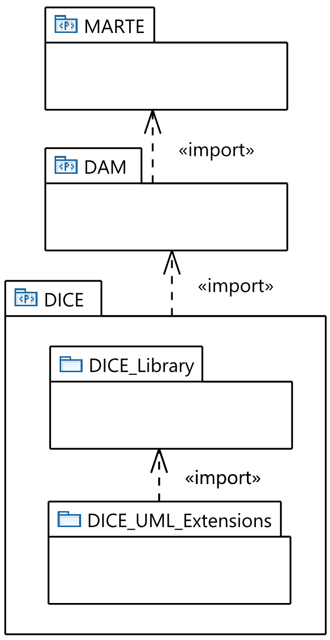

Figure 3: DICE Profile - High-level View

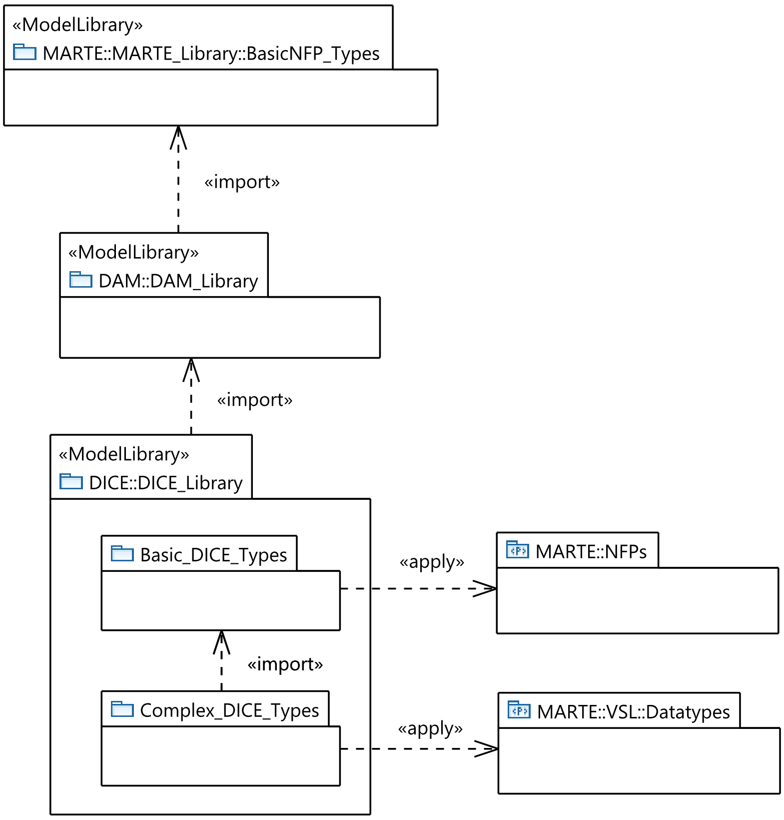

Figure 4: DICE Library

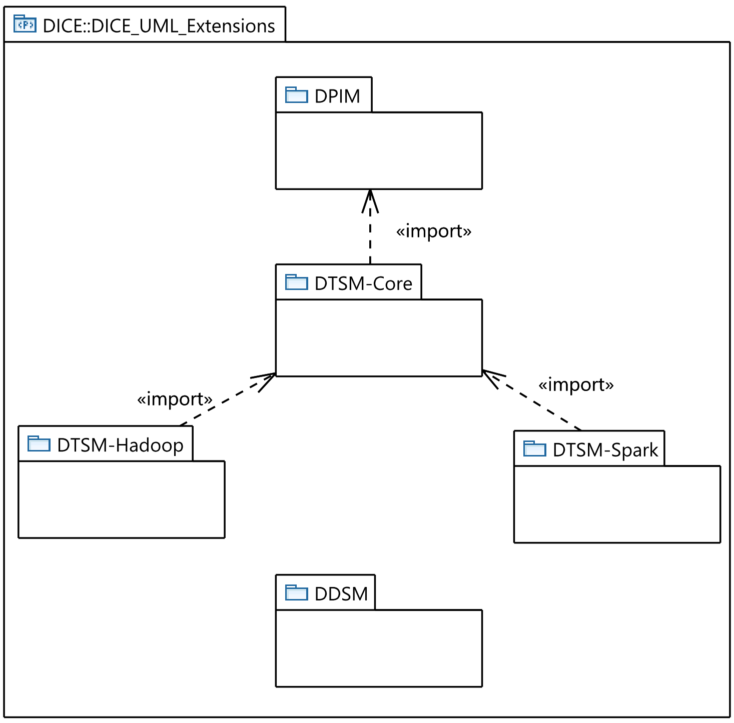

Figure 5: DICE UML Extensions

4 DPIM Logical Layer

DICE provides Software Architects with a set of core concepts, at the DPIM layer, to specify the fundamental architecture elements that constitute a Data-Intensivef application, i.e., during the *DIA Design* phase. Designers may use the identified core architecture elements to quickly put together the structural view of their Big-Data application, highlighting and tackling concerns such as data flow and essential high-level processing properties (e.g., rate, properties provided and required by every component, etc.) as well as key data processing needs (e.g., batch, streaming, etc.).

4.1 Metamodel

DPIM includes all concepts that are relevant for structuring a DIA. At the DPIM level we define the high level topology of the application and its QoS requirements. Elements of the DPIM meta-model fall into two categories:

1. Active DIA elements, which are shown by in light yellow in the bottom of the model (see Fig. 6);
2. Passive DIA elements, which are shown in white in the top of the model (see Fig. 6).

More in particular, the meta-model in Fig. 6 shows that DIA elements are essentially aggregates of two sets of components. Firstly, the "ComputationalNode", which is basically responsible for carrying out computational task like map, or reduce in MapReduce. One of important attributes of ComputationNode is "computationType" that shows the processing type of big-data i.e, batch processing or stream processing. The ComputationNode itself, further specializes into "SourceNode" and "Visualization" nodes. The SourceNode's role is to provide data for processing. In other word the SourceNode represents the source of data which are coming into application in order to being processed. The attribute "sourceType" further specifies the characteristics of source. The ultimate goal of a big-data application is to process the data that have high volume and velocity. So the SourceNode, and ComputationNode are in DPIM since there are the essential part of each and every DIA.The sourceNode is the entry point of data into application and the Computation is where data would process. Visualization here means to visualize the data to represent the knowledge more intuitively and effectively by using different graphs which are computed through Data-Intensive means. Even though the visualization of big-data itself could be done by a separate application, but here we considered visualization as specification of ComputationNode since ultimately the visualization is a data-intensive computation task. Another element which is also specification of ComputationNode is the FilterNode. Its role is to do any type of pre-processing and post-processing of data if needed.

The second key element in the DICE profile is the "StorageNode". As its name may suggest the StorageNode represents the element which is responsible for storing the data, either for long term or not. Moreover it is associated with "Channel" that represents the communication channel in application. The specification of Channel also shows the restrictions and constraints of a channel. It also specifies the characteristics related to transformation of data like information rate and taps. The concept of StorageNode in DPIM mainly corresponds with the "database" in the model1, in some case it could be "filesystem" also. The channel in DPIM is representation of "Governance and data Integration" in Model1 which mainly includes the technologies responsible for transferring the data, like message broker systems. The other elements in the model are "DataSpecification" and "QoSRequiredProperty", which are annotation stubs[1] for specification the type and format of data and the QoS for system and its evaluation respectively. Further details are available in Appendix A.

[1]Inherited from the MODACloudML notation (http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D4.2.1_MODACloudMLDevelopmentInitialVersion.pdf).

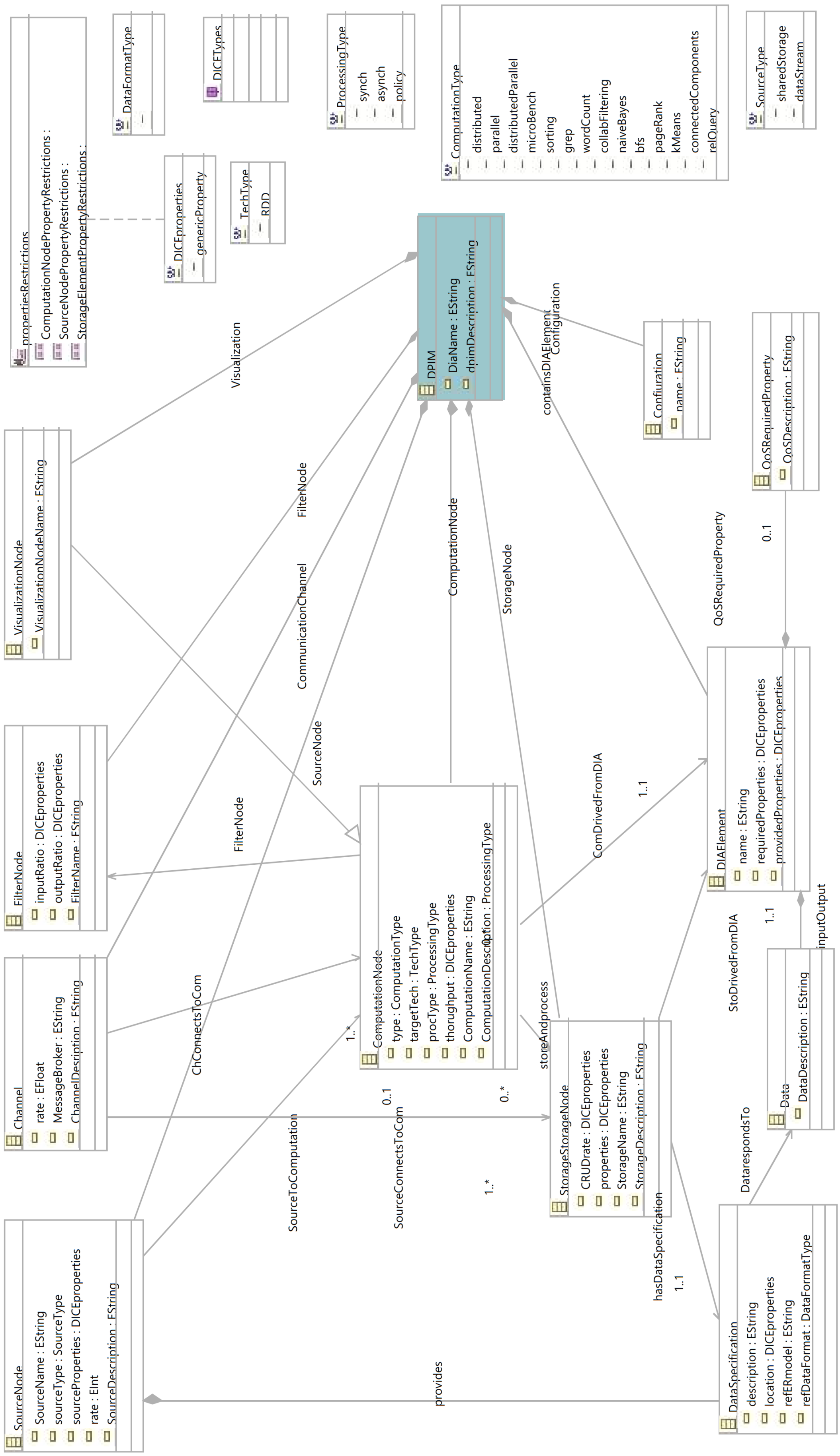

Figure 6: DICE DPIM metamodel

4.2 Profile

Table 1 summarizes the current list of stereotypes of the DICE Profile for the DPIM level.

Table 1: Stereotypes at DPIM level

Stereotype	**Description** (This stereotype is for model elements representing. . .)
`DiceComponent`	DIA components with throughput and maybe resource multiplicity.
`DiceFilterNode`	Filter nodes with input and output ratios.
`DiceSourceNode`	DIA components with a given storage volume and processing rate.
`DiceStorageResource`	DIA resources with resource multiplicity, data specification and processing rate.
`DiceChannel`	DIA channels that can be subject of failures and have error propagation rate.

5 DTSM Logical Layer

When all essential architecture elements are in place, by means of architectural reasoning in the DICE DPIM layer, DICE makes available ad-hoc model transformations that parse DPIM models and produce equipollent DTSM models where the specified data processing needs are exploded (if possible) into a possible configuration using appropriate technologies (e.g., Spark for streaming or Hadoop for batch). At this layer DICE provides architects and developers several key technological framework packages that they can evaluate as possible alternatives for *Technological Mapping* and *Logical Implementation*, that is, selecting the technological frameworks that map well with the problem at hand and implementing the needed processing logic for that framework. Once designers choose the appropriate technological alternative, DICE will provide model transformations that instantiate the alternative (if available) e.g., by instantiating pre-made, ad-hoc packages that contain: (a) framework elements needed to "link" the Data-Intensive application logic (e.g., through inheritance); (b) framework elements that contain (optional) configuration details (c) framework elements that represent deployable entities and nodes (e.g., Master Nodes and Resource Managers for Hadoop Map Reduce). Software Architects proceed by filling out any wanted configuration details to run the chosen frameworks, probably in collaboration with *Infrastructure Engineers*.

5.1 Metamodel

As previously introduced, at the DTSM layer the Data-Intensive application is elaborated with technology-specific packages. Several of the packages to be supported by DICE have been produced in the form of well-formed and validated meta-model package drafts. These drafts exist as working meta-models to be improved through action research with our industry partners. The technological packages that were currently drafted are: (1) Hadoop Map-Reduce 2; (2) Storm; (3) Spark; (4) Oryx 2. For example, see the Spark DTSM package[2] reported in Fig. 7. In essence, at the DTSM layer, designers use the pre-made technology-specific packages either with full defaults or configuring (at least) the following: (a) *WorkflowSpecification* - contains details necessary to configure the chosen framework according to ad-hoc execution, scheduling and access policies; (b) one or more *Tracker*(s) must be specified for the purpose of tracking job-progress and node-status; (c) a *Manager* node needs to be specified to regularly poll tracker nodes for the purpose of updating and steering their orchestration; (c) finally, *InputSplits* and *InputSplitSpec* must be produced to instruct the framework and the Big-Data application on how to produce input splits (i.e., processable blocks of coherent data) and what shall be the structure and semantics of said splits for further process. In addition, at the DTSM level, designers and developers elaborate on the desired behavioural specification for the data-intensive job, by "filling-in" the functions provided in the technology-specific packages, e.g., by inserting desired behaviour in the «`Map`» and «`Reduce`» constructs part of the Hadoop package within DICE.

5.2 Profile

Table 2 summarizes the current list of stereotypes of the DICE Profile for the `DTSM::Core` package. Further details about stereotypes for technologies are available in Appendix B.

Table 2: Stereotypes for the `DTSM::Core`

Stereotype	Description
`DiceComponent`	This stereotype inherits from one at the DPIM level and adds a function specification.
`DiceWkSpec`	This stereotype is for specifying DIA scenarios to which a performance or reliability analysis can be carried out.
`DiceSourceNode`	This stereotype is imported from the DPIM level.
`DiceStorageResource`	This stereotype inherits from one at the DPIM level and adds constraint and a management layer.

[2]More details on the Meta-Modeling Elements part of the various Technological Packages as well as the DICE Core Package are available in Appendix A

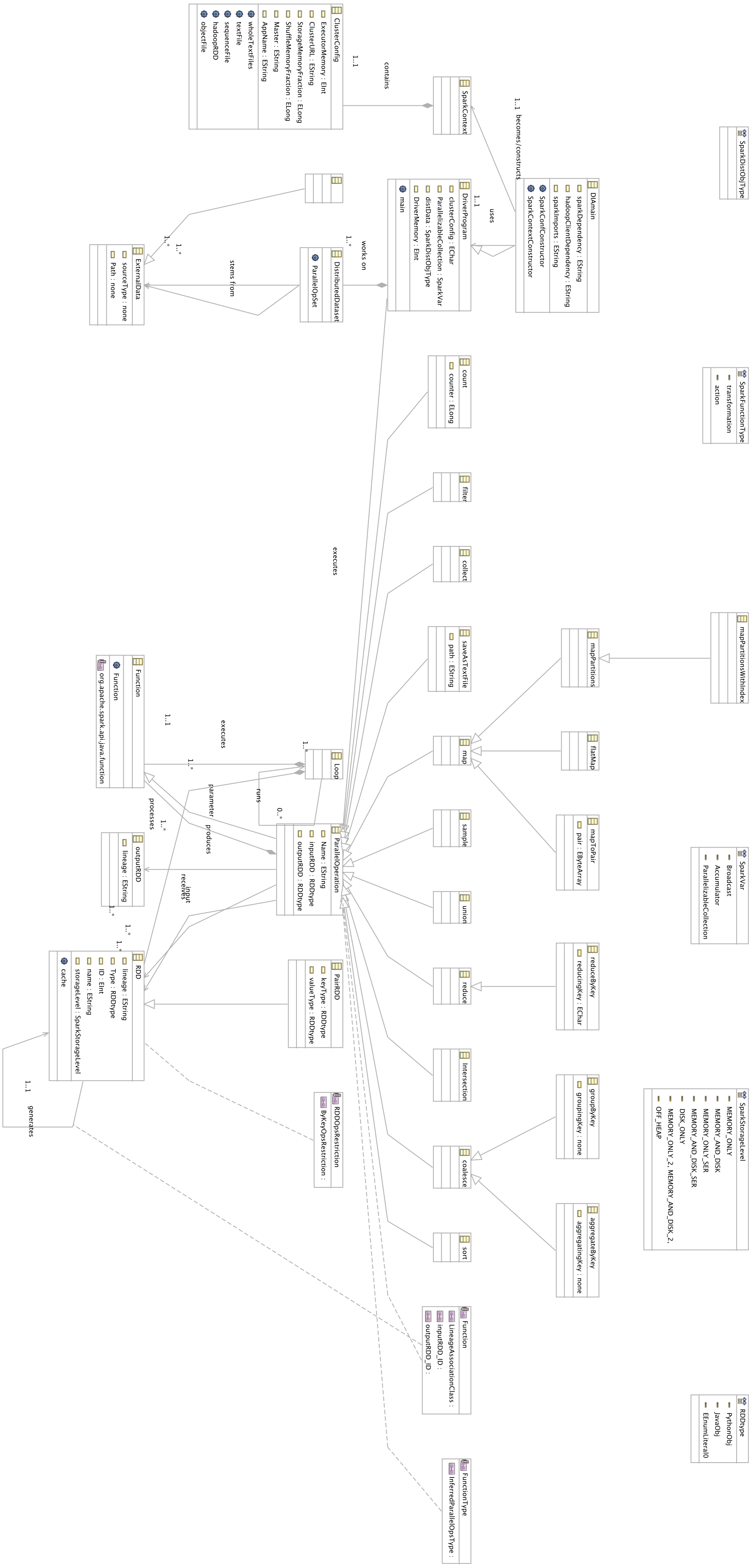

Figure 7: The DICE Spark Package, an Overview

6 Further Research Roadmap

Although we have achieved results almost beyond our own expectation, reaching a set of mature notations for the several technologies we plan to support in the DICE project requires still a significant amount of work.

First, in terms of Quality of Service (QoS) and Quality of Data (QoD) metrics, as previously mentioned, the DICE profile currently supports the specification of Quality required properties at the DPIM level (as envisioned in our requirements) but it does allow only partially to further elaborate said metrics into the parameters needed to verify/evaluate them. This support is actually only covered for the Apache Storm big data framework modelled within our specifications. Conversely, these parameters should be investigated extensively as part of further collaboration with remaining work-packages, namely WP3, WP4 and WP5. So far, we carried out some basic work to bring about some agreement on the set of high-level properties currently to be supported by the profile but, indeed, we are still working to elicit a complete set of said parameters to be implemented in an ancillary DTSM layer parameters library. We plan to elaborate said library either by: (a) investigating all possible parameters that can be elicited through DICE monitoring; (b) running empirical research with our fellow DICE partners; (c) reverse-engineer supported technological packages to understand key metrics to be supported.

Second, in terms of model evolvability and manipulability by means of model transformations, we currently developed a series of tentative technological transformation templates to be used as an inspiration for the development and implementation of the IDE. Said transformations are by no means complete and we are still gathering a complete overview for the expected transformation and manipulation requirements from our case-study owners. An important aspect that we have to study in detail concerns how to make sure that transformations enable an incremental modeling approach where the designer can move from the DPIM level down to the DTSM and DDSM levels and then back to DPIM for extending/modifying the DIA, without causing that his/her work on the lower levels goes lost.

Third, an important aspect to be tackled concerns the identification and modeling of privacy concerns which are key to the DICE tenants. This will be done based on the interaction with case study owners and with our legal consultant, Dott. Perri.

Stemming from the above discussion, in the future we plan to: (a) engage our case-study owners for the purpose of eliciting and further elaborating required QoS, QoD and privacy properties with connected parameters needed for verification; (b) engage our case-study owners for the purpose of gathering a complete overview of desired model manipulations; (c) draft an Eclipse IDE implementation and extension of the MARTE/DAM profile to support DICE models instantiation and manipulation; (d) evaluate said work with prototyping and industrial action research on our case studies.

7 Conclusions

In this document we have presented the current version of the DICE Metamodels and DICE Profile, which are the main outcomes for Tasks T2.1 and T2.2, respectively.

Table 3 summarizes the main achievements of this deliverable in terms of compliance with the initial set of requirements presented in Section 2. Note that some of the requirements are specified both at Profile level and at Methodological level. However, our analysis of compliance refers to the Profile level exclusively. In Table 3, the labels specifying the *Level of fulfillment* could be: (i) ✗ (unsupported: the requirement is not fulfilled by the current version); (ii) ✔ (partially-low supported: a few of the aspects of the requirement are fulfilled by the current version); (iii) ✔ (partially-high supported: most of the aspects of the requirement are fulfilled by the current version); and (iv) ✔ (supported: the requirement is fulfilled by the current version).

Table 3: Level of compliance of the current version with the initial set of requirements

Requirement	Title	Priority	Level of fulfillment
R2.1	DICE Methodological Paradigm	MUST	✗
R2.2	Abstraction Layer Origin	MUST	✔
R2.3	Relation with MARTE UML Profile	MUST	✔
R2.4	DICE Constraints Specification	MUST	✔
R2.5	DICE Profile Performance Annotations	MUST	✔
R2.6	DICE Profile Reliability Annotations	MUST	✔
R2.7	DICE Extension-Points	MUST	✔
R2.8	DICE Profile Main DIA Concerns - Flow and Behavior	MUST	✔
R2.9	DICE Topologies	MUST	✗
R2.10	DICE Profile Tech-Specific Constraints	MUST	✔
R2.11	DICE Profile Separation-of-Concerns	MUST	✔
R2.13	DICE Profile Data Structure	MUST	✔
R2.14	DICE Profile Data Communication	MUST	✔
R2.15	DICE Profile Sub-Structures	MUST	✔

7.1 Further Work

Tasks T2.1 and T2.2 will still produce an additional deliverable, D2.2, the *Design and quality abstractions - Final version* at month 24. For such deliverable we still need to: (i) introduce Storm and Oryx DICE Profiles at DTSM level; (ii) perform a complete validation of the DICE Profile; and (iii) address the following issues:

- Regarding requirement R2.1, this deliverable has not addressed the incremental specification.
- Regarding requirement R2.6, although some stereotypes already include the notion of failure, the DICE Profile needs to be reviewed for including reliability annotations compliant with DAM profile.
- Regarding requirement R2.7, it has been addressed at DTSM level but not at DDSM level.
- Regarding requirement R2.9, it needs to be addressed.
- Regarding requirement R2.10, it needs to be addressed for Storm and Oryx.
- Regarding requirement R2.15, currently this is supported by DAM.

Appendix A. Domain Metamodels

A.1 The `DiceDomainModel::DPIM` metamodel

Table 5: `DiceDomainModel::DPIM` data types

Name	Kind	Values or Description
`ComputationType`	Enumeration	`distributed`, `parallel`, `distributedParallel`, `microBench`, `sorting`, `grep`, `wordCount`, `collabFiltering`, `naiveBayes`, `bfs`, `pageRank`, `kMeans`, `connectedComponents`, `relQuery`
`TechType`	Enumeration	`RDD`
`SourceType`	Enumeration	`sharedStorage`, `dataStream`
`ProcessingType`	Enumeration	`synch`, `asynch`, `policy`
`DICEproperties`	Enumeration	`genericProperty`
`DataFormatType`	Enumeration	

Table 6: The `DiceDomainModel::DPIM` package

DICE `DiceDomainModel::DPIM` Metamodel Element	Description	Attributes
`DIA`	Represents a Data Intensive Application.	1. Compositions: • `contains`: `DIAElement`
`DIAElement`	An element of a Data Intensive Application. It can be a ComputationNode or a StorageNode.	1. Attributes: • `name`: `String` • `requiredProperties`: `DICEproperties` • `providedProperties`: `DICEproperties` 2. Compositions: • `hasToFullfill`: `QoSRequiredProperty` • `contains`: `DIAElement` • `input`: `Data` • `output`: `Data` 3. Associations: • `hasToFullfill`: `QoSRequiredProperty` • `contains`: `DIAElement` • `input`: `Data` • `output`: `Data`

`ComputationNode` inherits from: `DiceDomainModel::DPIM::DIAElement`, `DAM::System::Core::Component`	Represents an element of the application whose goal is to perform some computation.	1. Attributes: • `type`: `ComputationType` • `targetTech`: `TechType` • `procType`: `ProcessingType` • `thorughput`: `DICEproperties` 2. Associations: • `storeAndProcess`: `StorageNode`
`FilterNode` inherits from: `DiceDomainModel::DPIM::DIAElement`, `DiceDomainModel::DPIM::ComputationNode`	Represents a computation node that performs just some filtering on the application data.	1. Attributes: • `inputRatio`: `DICEproperties` • `outputRatio`: `DICEproperties`
`VisualizationNode` inherits from: `DiceDomainModel::DPIM::DIAElement`, `DiceDomainModel::DPIM::ComputationNode`	Represents a computation node whose goal is to perform data visualization.	
`SourceNode` inherits from: `DiceDomainModel::DPIM::DIAElement`, `DiceDomainModel::DPIM::ComputationNode`	This entity represents an element of the DIA acting as a data source at the DPIM layer.	1. Attributes: • `sourceType`: `SourceType` • `sourceProperties`: `DICEproperties` • `rate`: `double` 2. Associations: • `provides`: `DataSpecification`
`StorageNode` inherits from: `DiceDomainModel::DPIM::DIAElement`, `MARTE::GRM::ResourceCore::StorageResource`	Reseprents an element of the application whose goal is to store the application data, (i.e, a database or a file system).	1. Attributes: • `CRUDrate`: `DICEproperties` • `properties`: `DICEproperties` 2. Associations: • `responds_to`: `DataSpecification`
`Channel` inherits from: `DAM::System::Core::Connector`	Represents a communication channel between a ComputationNode and a StorageNode.	1. Attributes: • `rate`: `DICEproperties` 2. Associations: • `connectsOne`: `ComputationNode` • `connectsTwo`: `StorageNode`

`DataSpecification`	This entity represents data characteristics like the model and the format.	1. Attributes: • `description`: `String` • `size`: `int` • `location`: `DICEproperties` • `refERmodel`: `String` • `refDataFormat`: `DataFormatType`
`QoSRequiredProperty`	Represents the QoS constraints associated with an element of the Data Intensive Applicaion.	
`Data`	Represents the data that an element of the application can take in input and/or produce in output.	1. Associations: • `responds_to`: `DataSpecification`

A.2 The DiceDomainModel::DTSM::Core metamodel

Table 7: DiceDomainModel::DTSM::Core data types

Name	Kind	Values or Description
ConstraintsType	Enumeration	maxIteration
ComputationalNodeType	Enumeration	HadoopMR, Storm
ManagementLayerType	Enumeration	spark
JobScheduleType	Enumeration	runtime, speculative, redundant, fair, capacity
AccessScheduleType	Enumeration	
FunctionSpecType	Enumeration	map, reduce, combine, partition, report, collectOutput

Table 8: The DiceDomainModel::DTSM::Core package

DICE DiceDomainModel::DTSM::Core Metamodel Element	Description	Attributes
DIAElement inherits from: DiceDomainModel::DPIM::DIAElement	This entity represents a generic element of the DIA at the DTSM layer.	1. Attributes: • nodeTypeSpec: ComputationalNodeType
AnalyzableElement inherits from: DiceDomainModel::DTSM::Core::DIAElement	This entity represents an elemnt of the DIA application that have be analyzed in the sense that need to respect specific QoS requirements.	1. Attributes: • QoSRequiredProperty:
StorageNode inherits from: DiceDomainModel::DPIM::StorageNode	This entity represents a storage node in the DIA implemented by a specific storage technology at the DTSM layer.	1. Attributes: • nodeConstraints: String • managementLayer: ManagementLayerType 2. Associations: • affects: WorkflowSpecification
ComputationNode inherits from: DiceDomainModel::DTSM::Core::DIAElement, DiceDomainModel::DPIM::ComputationNode	This entity represents an element of the DIA performing a computation task by employing a specific data processing technology at the DTSM layer.	1. Attributes: • function: FunctionSpecType 2. Associations: • nestingAndReplication: ComputationNode

SourceNode inherits from: DiceDomainModel::DTSM::Core::DIAElement, DiceDomainModel::DPIM::SourceNode	This entity represents an element of the DIA acting as a data source at the DTSM layer.	1. Attributes: • name: String • type: SourceType 2. Associations: • loadInMemory: ChannelSpecification
WorkflowSpecification inherits from: DiceDomainModel::DTSM::Core::DIAElement, DAM::System::Core::Service	This entity represents an element of the DIA descrbing its workflow in terms of constraints affecting the storage and the computation nodes.	1. Attributes: • workflowConstraints: ConstraintsType • jobSchedule: JobScheduleType • blockAccessSchedule: AccessScheduleType 2. Associations: • restricts: ComputationNode
ChannelSpecification	This entity represents an element of the DIA responsible for the communication between components and the data trasnfer, implemented by a specific messaging technology at the DTSM layer.	1. Attributes: • rate: float • techSupport: TechType • size: float • policy:

A.3 The DiceDomainModel::DTSM::Hadoop metamodel

Table 9: DiceDomainModel::DTSM::Hadoop data types

Name	Kind	Values or Description
SplitEType	DataType	Split
connection	DataType	java.sql.connection
HadoopInputFormatEnum	Enumeration	TextInputFormat, FixedLengthInputFormat, SequenceFileInputFormat, KeyValueTextInputFormat, NLineInputFormat

Table 10: The DiceDomainModel::DTSM::Hadoop package

DICE DiceDomainModel::DTSM::Hadoop Metamodel Element	Description	Attributes
HadoopSpecificationModel inherits from: DiceDomainModel::DTSM::Core::WorkflowSpecification	This entity defines different configurations that are set by the developer and are required during the job execution. These information will be passed to the master node to define how the job should be scheduled.	1. Attributes: • mapperClass: String • reducerClass: String • jobName: String • combinerclass: String • numOfReduceTasks: int • inputFormat: String • outputKeyClass: String • outputValueClass: String • joinerClass: String • outputFormat: String • isJobSucceded: boolean 2. Compositions: • associated_to_HadoopMRrunner: HadoopMRrunner

`HadoopMRrunner`	This entity represents the application runner. After the application as been designed with all its components (Mapper, Reducer), this runner is responsible to effectively submit the application to the Hadoop runtime system.	1. Attributes: • `MapTaskReport`: `String` • `ReduceTaskRepor`: `String` • `JobID`: `int` • `clusterStatus`: `String` • `runningJobs`: `String` • `jobProgress`: `String` • `jobQueue`: `String` 2. Compositions: • `contains_RecordWriter`: `RecordWriter` • `contains_RecordReader`: `RecordReader` • `contains_DBaccessManager`: `DBaccessManager` • `contains_Scheduler`: `Scheduler` • `contains_Reducer`: `Reducer` • `contains_tester`: `Tester` • `contains_Joiner`: `Joiner` • `contains_mappr`: `Mapper`
`DBaccessManager`	This entity represents the driver the Hadoop application uses to access a data store. It allow to specify property like the url to the database and the username and password required to access it. Moreover through this class the application can directly operate on the databse.	1. Attributes: • `connection`: `String` • `initialised`: `boolean` • `isOracle`: `boolean` • `isMySQL`: `boolean` • `DB_URL`: `String` • `DRIVER_CLASS`: `String` • `server`: `String` • `Password`: `String` • `UserName`: `String` • `tableName`: `String` • `Conditions`: `String` • `OrderByFeildName`: `String` • `FieldNames`: `String`
`RecordWriter`	RecordWriter writes the output <key, value> pairs to an output file.	1. Attributes: • `finalSynch`: `boolean` • `out`: `String`

`RecordReader`	The record reader breaks the data into key/value pairs for input to the Mapper.	1. Attributes: • `startOffset`: `long` • `end`: `long` • `pos`: `long` • `fs`: `String` • `path`: `String` • `value`: `String` • `fileIn`: `String` • `reader`: `String` • `key`: `String` • `isInputSplitable`: `boolean` • `InputFormatType`: `HadoopInputFormatEnum` 2. Compositions: • `uses_InputSplitDataSpec`: `InputSplitDataSpec` • `uses_KeyValueGenerator`: `KeyValueGenerator`
`Mapper`	it is an implementation of map task that will be executed on some slave node in the deployed cluster, it will generate key-value pairs to be passed to reducers.	1. Attributes: • `mapper`: `String`
`Reducer`	it is an implementation of reduce task that will be executed on slave nodes, like map task. The key-value pairs made by reducers will be saved back into some storage like HDFS or an instance of a RDBMS.	1. Attributes: • `reducer`: `String`
`Tester`	This entity represents a generic suite of tests for a Mapper or a Reducer.	

`Scheduler`	The Scheduler in an Hadoop system is the component responsible assign key-value pairs produced by a Mapper to the proper Reducer. Each Scheduler is responsible for a given set of hosts (a cluster) and a give set of keys.	1. Attributes: • `USE`: `String` • `LOG`: `String` • `slotsPerHost`: `BigInteger` • `RemainingSplits`: `BigInteger` • `realSplits`: `Split` • `Splits`: `Split` • `host`: `String` 2. Associations: • `manages_host`: `Host` • `manages_split`: `Split`
`Joiner`	This entity represents a component used by the Hadoop application to join the output of a set of reducers.	1. Attributes: • `REDUCES_PER_HOST`: `int`
`Host`	This entity simply represents an host, like a virtual machine.	1. Attributes: • `hostName`: `String` • `splits`: `Split`
`Split`	This entity represent a piece of the application input data (i.e. a line in a file), that can be processed in parallel.	1. Attributes: • `filename`: `String` • `isAssigned`: `boolean` • `location`: `String`
`HadoopMRInputSpecs`	This entity represents the specification of the Hadoop application, like the path to the input text file to process.	1. Attributes: • `fileName`: `String`
`InputSplitDataSpec`	it gives the intuition to the developer about the structure of input files that should be read and written back into storage node i.e. HDFS data nodes.	1. Attributes: • `fileName`: `String` • `offSet`: `long` • `splitSize`: `long` • `maxSplitSize`: `long` • `MinSplitSize`: `long`

`KeyValueGenerator`	This entity represents an element of an Hadoop application able to generate key-value pairs.	1. Attributes: • `entry`: `String` • `Progress`: `float` 2. Compositions: • `generates`: `KeyValuePairs`
`KeyValuePairs`	This entity simply represents a key-value pairs produced by the Hadoop application.	1. Attributes: • `key`: `String` • `value`: `String`

A.4 The DiceDomainModel::DTSM::Oryx metamodel

Table 11: The DiceDomainModel::DTSM::Oryx package

DICE DiceDomainModel::DTSM::Oryx Metamodel Element	Description	Attributes
DIA	This entity represents the root of the Oryx 2 DIA.	1. Attributes: • type: String • DIADescription: String • runScript: String • computeClassPath: String • id: String 2. Compositions: • DIASer: ServingLayer • DIABa: BatchLayer • DIASp: SpeedLayer
Kafka	This entity represents an instance of Kafka that is used by the Oryx 2 system as the data transport layer, which moves data between layers of the Lambda architecture and receives input from external sources	1. Attributes: • type: String • KafkaDescription: String • BrokerURL: String 2. Compositions: • HasUpdateTopic: updateTopic • KafkaHasInputTopic: inputTopic
Zookeeper	This entity represents an instance of Zookeeper that is used the Oryx 2 framework.	1. Attributes: • type: String • ZookeeperDescription: String • zkServers: String

APISpecification	This entity represents the specification of the APIs exposed by the serving layer.	1. Attributes: • `UserName`: `String` • `Password`: `String` • `servingLayerPort`: `String` • `APIDescription`: `String` • `keystoreFile`: `String` • `keystorePassword`: `String` • `readOnly`: `boolean` • `contextPath`: `String` • `typeApi`: `String`
YarnSpecification	This entity represents the configuration of the YARN cluster on top of which the BatchLayer run.	1. Attributes: • `type`: `String` • `NoOfInstance`: `int` • `Cores`: `int` • `YarnDescription`: `String`
MlSpecification	This entity represents the specification of the Machine Learning algorithm to be executed, allowing to specify parameters like the degree of parallelism and the test fraction of the input dataset.	1. Attributes: • `TestFraction`: `int` • `Candidate`: `int` • `Parallelism`: `int` • `MISpecificationDescription`: `String` • `typeMl`: `String`
StorgeSpecification	This entity represent the storage system on which the Batch layer store results. It is implemented in Oryx 2 as an HDFS.	1. Attributes: • `type`: `String` • `InputDirectory`: `String` • `OutputDirectory`: `String` • `StorageDescription`: `String` • `Rate`: `int` • `SourceProperties`: `String`
ConfSpecification inherits from: `DiceDomainModel::DTSM::Oryx::APISpecification`, `DiceDomainModel::DTSM::Oryx::YarnSpecification`, `DiceDomainModel::DTSM::Oryx::MlSpecification`	This entity represents a generic system configuration which can be extended according to specific systems employed in the Oryx 2 architecture.	1. Attributes: • `ConfigurationDescription`: `String`

`SparkStream`	This entity represents a Spark Streaming application. In Oryx 2 both the Speed layer and the Batch layer are implemented as Spark Streaming applications.	1. Attributes: • `type`: `String` • `IntervalBtwnComputation`: `String` • `master`: `String` • `NumberofExecutors`: `String` • `ExecutorCore`: `String` • `ExecutorMemory`: `String` • `HeapSize`: `String` • `DynamicAlloc`: `boolean`
`ServingLayer`	The serving layer listens for model and model updates on the update topic Kafka Topic. It maintains model state in memory. It exposes REST APIs for queryign the model in memory. Each API may also accept new data and write it to Kafka where it can be seen by the Speed and Batch layers.	1. Attributes: • `type`: `String` • `ModelManagerClass`: `String` • `ApplicationResources`: `String` • `memory`: `String` • `ServinLayerDescription`: `String` • `minModelLoadFraction`: `String` 2. Compositions: • `SerUseKf`: `Kafka` • `SerManageMI`: `MlSpecification` • `SpeAPI`: `APISpecification` • `SpeYarn`: `YarnSpecification`
`SpeedLayer`	This entity represents the Speed layer of the Lambda architecture and is implemented using Spark Streaming. It periodically loads a new model from the update topic and continually produces model updates. These are put back onto the update topic too.	1. Attributes: • `modelManagerClass`: `String` • `type`: `String` • `UiPort`: `String` • `DynamicAlloc`: `String` • `minModelLoadFraction`: `String` • `SpeedLayerDescription`: `String` 2. Compositions: • `SpeUseMI`: `MlSpecification` • `SpeUseKaf`: `Kafka` • `SpeedhasSparkStream`: `SparkStream`

`BatchLayer`	The batch layer is implemented as a Spark Streaming process on a Hadoop cluster, which reads data from the input Kafka topic. The Streaming process necessarily has a very long period – hours or even a day. It uses Spark to save the current window of data to HDFS, and then combine with all historical data on HDFS, and initiate building of a new result. The result is written to HDFS, and, also published to a Kafka update topic.	1. Attributes: • `updateClass`: `String` • `type`: `String` • `HDFSbaseURL`: `String` • `UiPort`: `String` • `BatchLayerDescription`: `String` 2. Compositions: • `BatchHasStorage`: `StorgeSpecification` • `BatchUseMI`: `MlSpecification` • `BatchUseKaf`: `Kafka` • `BatchhasSparkStream`: `SparkStream`
`inputTopic`	This entity represent the Kafka Topic used for new inputs of the system. The serving layer post new inputs to this Topic which are consumed by the Speed and Batch layers.	1. Attributes: • `type`: `String` • `KafkaConsumerDescription`: `String` • `name`: `String` • `NumberofPartitions`: `String` • `retentionTime`: `String` • `replicationValue`: `String` • `maxMessageSize`: `String` 2. Compositions: • `InputTopichasMaster`: `Zookeeper`
`updateTopic`	This entity represent the Kafka Topic used for update the model in order to reflect new input. The Serving layer consumes the model and the model updates to answer to the user queries, the Speed layer periodically consume the current model and produce a model update, while the Batch layer at each execution produce a new model.	1. Attributes: • `type`: `String` • `KafkaProducerDescription`: `String` • `name`: `String` • `NumberOfPartitions`: `String` • `retentionTime`: `String` • `replicationValue`: `String` • `maxMessageSize`: `String` 2. Compositions: • `UpdatTopichasMaster`: `Zookeeper`

A.5 The DiceDomainModel::DTSM::Spark metamodel

Table 12: DiceDomainModel::DTSM::Spark data types

Name	Kind	Values or Description
SparkDistObjType	Enumeration	
SparkFunctionType	Enumeration	transformation, action
SparkVar	Enumeration	Broadcast, Accumulator, ParallelizableCollection
SparkStorageLevel	Enumeration	MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, MEMORY_ONLY_2, MEMORY_AND_DISK_2,, OFF_HEAP
RDDtype	Enumeration	PythonObj, JavaObj, EEnumLiteral0

Table 13: The DiceDomainModel::DTSM::Spark package

DICE DiceDomainModel::DTSM::Spark **Metamodel Element**	**Description**	**Attributes**
DIAmain inherits from: DiceDomainModel::DTSM::Spark::DriverProgram	This entity represents the root of the Spark application and has an associated DriverProgram to execute.	1. Attributes: • sparkDependency: String • hadoopClientDependency: String • sparkImports: String 2. Associations: • constructs: SparkContext • uses: DriverProgram
DistributedDataset	This entity represents the dataset the Spark application have to process.	1. Associations: • : ExternalData • stems from: ExternalData

ParallelOperation inherits from: DiceDomainModel::DTSM::Spark::Function	A generic operation performing some trasformation in paralled on RDDs.	1. Attributes: • Name: String • inputRDD: RDDtype • outputRDD: RDDtype 2. Compositions: • parameter: Function • receives: RDD • produces: outputRDD • input: RDD 3. Associations: • parameter: Function • receives: RDD • produces: outputRDD • input: RDD
ExternalData	This entity represents an external data source from which RDDs can be created.	1. Attributes: • sourceType: • Path:
count inherits from: DiceDomainModel::DTSM::Spark::Function, DiceDomainModel::DTSM::Spark::ParallelOperation	Return the number of elements in an RDD.	1. Attributes: • counter: long
filter inherits from: DiceDomainModel::DTSM::Spark::Function, DiceDomainModel::DTSM::Spark::ParallelOperation	Return a new dataset formed by selecting those elements of the source RDD on which a given function returns true.	
collect inherits from: DiceDomainModel::DTSM::Spark::Function, DiceDomainModel::DTSM::Spark::ParallelOperation	Return all the elements of the dataset as an array.	
flatMap inherits from: DiceDomainModel::DTSM::Spark::Function, DiceDomainModel::DTSM::Spark::ParallelOperation, DiceDomainModel::DTSM::Spark::map	Similar to map, but each input item can be mapped to 0 or more output items instead of just a single item.	
mapToPair inherits from: DiceDomainModel::DTSM::Spark::Function, DiceDomainModel::DTSM::Spark::ParallelOperation, DiceDomainModel::DTSM::Spark::map	Return a new RDD by applying a function to all elements of this RDD.	1. Attributes: • pair: byte[]

`reduceByKey` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`, `DiceDomainModel::DTSM::Spark::reduce`	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using a given reduce function	1. Attributes: • `reducingKey`: `char`
`saveAsTextFile` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	Write the elements of the dataset as a text file (or set of text files) in a given directory in a filesystem	1. Attributes: • `path`: `String`
`map` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	Return a new RDD formed by passing each element of the source RDD through a function func.	
`reduce` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	This operation aggregates the elements of the RDD on which it is execute using a given reduce function.	
`DriverProgram`	The DriverProgram is the main application that declares the transformations and actions on RDDs using a instance of the SparkContext and submits such requests to the master which manages the cluster scheduling tasks among Workers and provides the SparkContext.	1. Attributes: • `clusterConfig`: `char` • `ParallelizableCollection`: `SparkVar` • `distData`: `SparkDistObjType` • `DriverMemory`: `int` 2. Compositions: • `executes`: `ParallelOperation` • `works on`: `DistributedDataset` 3. Associations: • `executes`: `ParallelOperation` • `works on`: `DistributedDataset`
`SparkContext`	The SparkContext is the main entry point for Spark functionality. A SparkContext represents the connection to a Spark cluster. It can be used for example to create RDDs from a datastore table (i.e. a Cassandra table).	1. Compositions: • `contains`: `ClusterConfig`

`Function`	The Spark Function is an interface representing a generic operation provided by Spark appliable on RDDs.	
`coalesce` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	This Spark operation decreases the number of partitions in an input RDD.	
`groupByKey` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`, `DiceDomainModel::DTSM::Spark::coalesce`	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.	1. Attributes: • `groupingKey`:
`aggregateByKey` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`, `DiceDomainModel::DTSM::Spark::coalesce`	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using a given combine functions.	1. Attributes: • `aggregatingKey`:
`mapPartitions` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`, `DiceDomainModel::DTSM::Spark::map`	Similar to map, but runs separately on each partition (block) of the RDD.	
`mapPartitionsWithIndex` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`, `DiceDomainModel::DTSM::Spark::map`, `DiceDomainModel::DTSM::Spark::mapPartitions`	Similar to mapPartitions, but also provides the executed function with an integer value representing the index of the partition	
`sample` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	Sample a fraction of the input RDD, with or without replacement, using a given random number generator seed.	
`union` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	Return a new dataset that contains the union of the elements in the RDD on which it is called and the argument RDD.	

Intersection inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	Return a new dataset that contains the intersection of the elements in the RDD on which it is called and the argument RDD.	
RDD	This entity represend and RDD (Resilient Distributed Dataset) and is the primary abstraction within the Spark framework. An RDD is a fault-tolerant, possibly distributed, set of items that can be processed in parallel. They can be for example generated from data stored in HDFS.	1. Attributes: • `lineage`: `String` • `Type`: `RDDtype` • `ID`: `int` • `name`: `String` • `storageLevel`: `SparkStorageLevel` 2. Associations: • `generates`: `RDD`
outputRDD	This entity represents the output RDD of the Spark application that can be stored.	1. Attributes: • `lineage`: `String`
ClusterConfig	This entity represents the configuration of the Spark cluster managed by the Master, like the number of nodes and characteristics of each node.	1. Attributes: • `ExecutorMemory`: `int` • `ClusterURL`: `String` • `StorageMemoryFraction`: `long` • `ShuffleMemoryFraction`: `long` • `Master`: `String` • `AppName`: `String`
Loop	This entity simply represent a loop executed by the Spark application.	1. Compositions: • `executes`: `Function` • `processes`: `RDD` • `runs`: `Loop` 2. Associations: • `executes`: `Function` • `processes`: `RDD` • `runs`: `Loop`
PairRDD inherits from: `DiceDomainModel::DTSM::Spark::RDD`	RDD that are just (key, value) pairs, for which Spark provides extra functions.	1. Attributes: • `keyType`: `RDDtype` • `valueType`: `RDDtype`

`sort` inherits from: `DiceDomainModel::DTSM::Spark::Function`, `DiceDomainModel::DTSM::Spark::ParallelOperation`	When this operation is called on a dataset of (K, V) pairs where, it returns a dataset of (K, V) pairs sorted by keys in ascending or descending order	

A.6 The DiceDomainModel::DTSM::Storm metamodel

Table 14: DiceDomainModel::DTSM::Storm data types

Name	Kind	Values or Description
BoltApi	Enumeration	prepare, execute, OPFields, Cleanup, Config
Operation	Enumeration	name, params, type, body
Property	Enumeration	name, type
SpoutApi	Enumeration	Open, NextTuple, Ack, Fail, OPFields
StormOpMode	Enumeration	local, remote

Table 15: The DiceDomainModel::DTSM::Storm package

DICE DiceDomainModel::DTSM::Storm Metamodel Element	Description	Attributes
DIAmain	This entity represents the root of the Stom application.	1. Attributes: • opMode: StormOpMode 2. Associations: • clusterManager: Nimbus • dependsOn: Zookeeper • clusteredVia: Supervisor • becomes: Topology • becomes: TopologyConfiguration
DIAStorage	This entity represents a storage element of the application, implemented by a specific technology, in which specific Bolt can write their results.	
DIAFilter	This entity represents an element of the Storm application that apply some preliminary filtering over streams produced by give data sources.	1. Associations: • preprocesses: DIASource
DIASource	This entity represents an external data source for the Storm application, from which specific Spouts can read input stream.	

StormSpecificationModel		1. Compositions: • aggregates: PackageDeclaration
Topology inherits from: DiceDomainModel::DTSM::Storm::StormSpecificationModel	This entity embeed the logic of the Storm application. It represents the application cointnous workflow over the input streams of tuples by means of a DAG (Directed Acyclic Graph) composed by Bolts and Spouts.	1. Attributes: • build: String • Reliable: boolean • name: String 2. Compositions: • logicalSpecification: Component • uses: TopologyConfiguration • crossFunctionalProcessing: Topology 3. Associations: • logicalSpecification: Component • uses: TopologyConfiguration • crossFunctionalProcessing: Topology
Component	This entity represents a generic element of a Storm Topology either a Spout or a Bolt.	1. Attributes: • name: String
PackageDeclaration inherits from: DiceDomainModel::DTSM::Storm::StormSpecificationModel	This represents the package declaration section of the Storm application.	
Spout inherits from: DiceDomainModel::DTSM::Storm::StormSpecificationModel, DiceDomainModel::DTSM::Storm::Topology, DiceDomainModel::DTSM::Storm::Component	This entity represents a Spout in the Storm topology, or an element that reads tuples from external data source and emits them into the topology.	1. Attributes: • inputSource: String 2. Associations: • processedBy: Bolt • readsFrom: DIASource

`Bolt` inherits from: `DiceDomainModel::DTSM::Storm::StormSpecificationModel`, `DiceDomainModel::DTSM::Storm::Topology`, `DiceDomainModel::DTSM::Storm::Component`	All processing in topologies is done in bolts. Bolts can do anything from filtering, functions, aggregations, joins, talking to databases, and more. This entity represents a Bolt in a Storm topology, or an element doing some processing on the input tuples. Kind of processing tasks can be joins on over two streams of tuples or an aggregation over a stream.	1. Attributes: • `inputMessage`: `String` • `outputMessage`: `String` 2. Associations: • `furtherProcessing`: `Bolt` • `storesIn`: `DIAStorage`
`TopologyBuilder`	This elements allow to build Storm Topologies by exposing suitable APIs.	1. Attributes: • `name`: `String` 2. Associations: • `builds`: `Topology` • `buildsSpouts`: `AddSpout`
`AddBolt` inherits from: `DiceDomainModel::DTSM::Storm::TopologyBuilder`	This element represents the operation of adding a Bolt in a Topology provided by a TopologyBuilder.	
`AddSpout` inherits from: `DiceDomainModel::DTSM::Storm::TopologyBuilder`	This element represents the operation of adding a Bolt in a Topology provided by a TopologyBuilder.	

`TopologyConfiguration`	This entity represents the topology specification, which has to be employed by the Stom application.	1. Attributes: • `parameter`: `String` • `path`: `String` • `maxSpout`: `int` • `ZookeeperConnectionTimeout`: `double` 2. Compositions: • `feedsInto`: `Nimbus` • `specifies`: `Task` • `specifies`: `Executor` • `specifies`: `Worker` • `specifies`: `Supervisor` • `specifies`: `Zookeeper` • `specifies`: `Nimbus` 3. Associations: • `feedsInto`: `Nimbus` • `specifies`: `Task` • `specifies`: `Executor` • `specifies`: `Worker` • `specifies`: `Supervisor` • `specifies`: `Zookeeper` • `specifies`: `Nimbus`
`Nimbus`	This represents the nimbus daemon running on the master node responsible for managing the cluster of slave nodes.	1. Attributes: • `stormFW`: `String` • `UI`: `String`
`Zookeeper`	This entity represents an instance of Zookeeper that is used the Oryx 2 framework.	1. Associations: • `fileManagement`: `Nimbus`
`Supervisor`	The Supervisor is a daemon responsible for starting and stopping Worker processes on a specific machine.	1. Attributes: • `stormFW`: `String` 2. Associations: • `isPartOf`: `Supervisor` • `isManagedBy`: `Nimbus` • `reference`: `Zookeeper` • `has`: `Worker`

`Task`	A Task represents a generic operation that need to be run and which is assigned to an Executor. It can be either the execution of a Bolt or of a Spout.	1. Attributes: • `replicationFactor`: `int` 2. Associations: • `instanceOf`: `Component`
`Executor`	An Executor represents a thread run by a Worker and executing in parallel one or more instances of the same specific Task assigned to the Executor.	1. Attributes: • `replicationFactor`: `int` 2. Associations: • `has`: `Task` • `taskConsistency`: `Topology`
`Worker`	A Worker represents a process within the Storm system which can run multiple Executors in parallel on top of a JVM and inside one virtual machine of the Storm cluster.	1. Attributes: • `ReplicationFactor`: `int` 2. Associations: • `has`: `Executor`

Appendix B. Profile mappings

Next we detail the mapping between the concepts of the DICE domain metamodels and the DICE profile. The engineer only needs to use those DICE tags that are useful for him/her to describe the UML model element at hand. Note that unqualified classifier names belong to the packages being described in the corresponding section.

B.1 Mapping the DiceDomainModel::DPIM metamodel onto DICE Profile: The DICE::DICE_UML_Extensions::DPIM package

Table 16: The DICE::DICE_UML_Extensions::DPIM package

DICE DPIM Metamodel Element	**DICE Stereotype**	**Applicable to**	**DICE Tags**
DIA	**Comment:** This class represents the model itself, then it does not map into a Profile element.		
DIAElement	**Comment:** This is an abstract class, then it does not map into a Profile element.		
ComputationNode	«DiceComponent» inherits from «DAM::DAM_UML_Extensions::System::Core::DaComponent»	From «DAM::DAM_UML_Extensions::System::Core::DaComponent» and supertypes (e.g. «MARTE::GRM::Resource»): UML::Classes::Kernel::Property, UML::Classes::Kernel::InstanceSpecification, UML::Classes::Kernel::Classifier, UML::Interaction::BasicInteractions::Lifeline, UML::CompositeStructures::InternalStructures::ConnectableElement	New tags: • throughput: NFP_Frequency • type: ComputationType • targetTech: TechType • procType: ProcessingType Inherited tags of interest: • isActive: bool • resMult: int
FilterNode	«DiceFilterNode» inherits from «DiceComponent»	All from «MARTE::GRM::Resource» (see «DiceComponent»).	New tags: • inputRatio: NFP_Frequency • outputRatio: NFP_Frequency
VisualizationNode	**Comment:** This class does not declare additional attributes, thus, an stereotype is not needed.		
SourceNode	«DiceSourceNode» inherits from «DiceComponent»	All from «MARTE::GRM::Resource» (see «DiceComponent»).	New tags: • store: DiceDataVolume • provides: DiceDataSpecification • sourceType: SourceType • rate: NFP_Frequency

StorageNode	«DiceStorageResource» inherits from «MARTE::GRM::StorageResource»	From «MARTE::GRM::StorageResource» and supertypes: UML::Classes::Kernel::Property, UML::Classes::Kernel::InstanceSpecification, UML::Classes::Kernel::Classifier, UML::Interaction::BasicInteractions::Lifeline, UML::CompositeStructures::InternalStructures::ConnectableElement	New tags: • respondsTo: DiceDataSpecification; • CRUDrate: NFP_Frequency Inherited tags of interest: • resMult: int • elementSize: int
Channel	«DiceChannel» inherits from «DAM::DAM_UML_Extensions::System::Core::DaConnector»	From «DAM::DAM_UML_Extensions::System::Core::DaConnector»: UML::Classes::Kernel::Association, UML::Classes::Dependencies::Dependency, UML::Components::BasicComponents::Connector, UML::Interactions::BasicInteractions::Message, UML::UseCases::Include, UML::UseCases::Extend, UML::CompositeStructure::InvocationActions::InvocationAction	New tags: • rate: NFP_Frequency • messageBroker: String • channelDescription: DiceChannelSpecification Inherited tags of interest: • coupling: NFP_Real[*] • failure: DaFailure[*] • errorProp: DaErrorPropagation[*]
DataSpecification	**Comment:** DataSpecification maps to a DICE complex type. See package DICE::DICE_Library::Complex_DICE_Types.		
QoSRequiredProperty	**Comment:** This is the definition of a MARTE NFP with source=req. This class is aggregated to DIAElement. Each DIAElement that needs a NFP definition should have its corresponding tag. Therefore, it is not mapped to a DICE stereotype.		
Data	**Comment:** Does not map onto a stereotype.		

B.2 Mapping the DiceDomainModel::DTSM::Core metamodel onto DICE Profile: The DICE::DICE_UML_Extensions::DTSM::Core package

Table 17: The DICE::DICE_UML_Extensions::DTSM::Core package

DICE DTSM::Core **Metamodel Element**	**DICE Stereotype**	**Applicable to**	**DICE Tags**
DIAElement	**Comment:** This is an abstract class, and in this initial proposal does not map into a Profile element. However, its attribute nodeTypeSpec still needs to be considered in a refactoring.		
AnalyzableElement	**Comment:** This class is for defining QoS properties (MTTF, MTTR, etc). They are already defined in MARTE and DAM as NFPs. Therefore, it does not map into a Profile element.		
StorageNode	«DiceStorageResource» inherits from «DICE::DICE_UML_Extensions::DPIM::DiceStorageResource»	See «DICE::DICE_UML_Extensions::DPIM::DiceStorageResource»	New tags: • nodeConstraints: String[*] • managementLayer: ManagementLayerType
	Comment: The affects association is not dealt by the Profile		
ComputationNode	«DiceComponent» inherits from «DICE::DICE_UML_Extensions::DPIM::DiceComponent»	See «DICE::DICE_UML_Extensions::DPIM::DiceComponent»	New tags: • function: FunctionSpecType
	Comment: The nestingAndReplication recursive association is not dealt by the Profile.		
SourceNode	«DiceSourceNode» imported from «DICE::DICE_UML_Extensions::DPIM::DiceSourceNode»	See «DICE::DICE_UML_Extensions::DPIM::DiceSourceNode»	
	Comment: A new stereotype is not needed since attributes, name and type are already present in the superclass.		
WorkflowSpecification	«DiceWkSpec» inherits from «DAM::DAM_UML_Extensions::System::Core::DaService»	Inherited from «DAM::DAM_UML_Extensions::System::Core::DaService» and supertypes (e.g., «MARTE::MARTE_AnalysisModel::GQAM::GaScenario»): UML::Classes::Kernel::NamedElement, UML::Actions::Action, UML::CommonBehaviors::Behavior, UML::Interactions::BasicInteractions::Message	New tags: • wkConstraints: ConstraintsType[*] • jobSchedule: JobSchedule • bASchedule: AccessSchedule
	Comment: The restricts association is not dealt by the Profile.		
ChannelSpecification	**Comment:** ChannelSpecification maps to a DICE complex type. See package DICE::DICE_Library::Complex_DICE_Types.		

B.3 Mapping the DiceDomainModel::DTSM::Hadoop metamodel onto DICE Profile: The DICE::DICE_UML_Extensions::DTSM::Hadoop package

Table 18: The DICE::DICE_UML_Extensions::DTSM::Hadoop package

DICE DTSM::Hadoop Metamodel Element	DICE Stereotype	Applicable to	DICE Tags
HadoopSpecificationModel	«DiceHadoopSpec» inherits from «DICE::DICE_UML_Extensions::DTSM::Core::DiceWkSpec»	See «DICE::DICE_UML_Extensions::DTSM::Core::DiceWkSpec»	New tags: • mapperClass: String • reducerClass: String • jobName: String • combinerclass: String • numOfReduceTasks: Integer • inputFormat: String • outputKeyClass: String • outputValueClass: String • joinerClass: String • outputFormat: String • isJobSucceded: boolean
HadoopMRrunner	«DiceHadoopRunner»	UML::Classes::Kernel::InstanceSpecification, UML::Classes::Kernel::Classifier, UML::Interaction::BasicInteractions::Lifeline, UML::CompositeStructures::InternalStructures::ConnectableElement	New tags: • rapTaskReport: String • reduceTaskReport: String • jobID: Integer • clusterStatus: String • runningJobs: String • jobProgress: String • jobQueue: String
	Comment: This stereotype may be useful only for informative purposes		

DBaccessManager	«DiceHadoopDBMngr»	UML::Classes::Kernel:: InstanceSpecification, UML::Classes::Kernel:: Classifier, UML::Interaction:: BasicInteractions:: Lifeline, UML::CompositeStructures:: InternalStructures:: ConnectableElement	New tags: • connection: String • initialised: Boolean • isOracle: Boolean • isMySQL: Boolean • dbUrl: String • driverClass: String • server: String • password: String • userName: String • tableName: String • conditions: String • orderByFieldName: String • FieldNames: String
	Comment: This stereotype may be useful only for informative purposes		
RecordWriter	«DiceHadoopRWriter» inherits from «DICE::DICE_UML_Extensions:: DTSM::Core::DiceStorageResource»	See «DICE:: DICE_UML_Extensions::DTSM:: Core::DiceStorageResource»	
RecordReader	«DiceHadoopRReader» inherits from «DICE::DICE_UML_Extensions:: DTSM::Core::DiceSourceNode»	See «DICE:: DICE_UML_Extensions::DTSM:: Core::DiceSourceNode»	
Mapper, Reducer, Tester	«DiceHadoopMROperation» inherits from «MARTE::MARTE_AnalysisModel:: GQAM::GaStep»	Inherited from «MARTE:: MARTE_AnalysisModel::GQAM:: GaStep» and supertypes: UML::Classes::Kernel:: NamedElement, UML::Actions::Action, UML:: CommonBehaviors::Behavior, UML::Interactions:: BasicInteractions::Message	
	Comment: «DiceHadoopMROperation» may inherit from a specific stereotype – not declared yet – in «DICE::DICE_UML_Extensions::DTSM::Core» to represent operations		
KeyValuePair	**Comment:** This class is uninteresting from the profile point of view, and does not require a stereotype		
InputSplitDataSpec	**Comment:** This class is uninteresting from the profile point of view, and does not require a stereotype		

B.4 Mapping the DiceDomainModel::DTSM::Spark metamodel onto DICE Profile: The DICE::DICE_UML_Extensions::DTSM::Spark package

Table 19: The DICE::DICE_UML_Extensions::DTSM::Spark package

DICE DTSM::Spark Metamodel Element	DICE Stereotype	Applicable to	DICE Tags
DriverProgram	«DiceSparkSpec» inherits from «DICE::DICE_UML_Extensions::DTSM::Core::DiceWkSpec»	See «DICE::DICE_UML_Extensions::DTSM::Core::DiceWkSpec»	
	Comment: «DiceSparkSpec» is used just for informative purposes. Time specification is done in «DiceSparkOperation».		
ParallelOperation	«DiceSparkOperation» inherits from «MARTE::MARTE_AnalysisModel::GQAM::GaStep»	Inherited from «MARTE::MARTE_AnalysisModel::GQAM::GaStep» and supertypes: UML::Classes::Kernel::NamedElement, UML::Actions::Action, UML::CommonBehaviors::Behavior, UML::Interactions::BasicInteractions::Message	New tags: • kind: SparkOperationKind
	Comment: The kind tag is just for informative purpose, and is uninteresting for performance or reliability analysis. «ParallelOperation» may inherit from a specific stereotype – not declared yet – in «DICE::UML_Extensions:DTSM::Core» to represent operations.		
RDD	«DiceSparkRDDataSet» inherits from «DICE::DICE_UML_Extensions::Core::DiceSourceNode»	See «DICE::DICE_UML_Extensions::Core::DiceSourceNode»	

B.5 DICE model library

The DICE model library contains basic and complex types that are used by the DICE UML extensions.

B.5.1 The DICE::DICE_Library::Basic_DICE_Types package

Table 20: The DICE::DICE_Library::Basic_DICE_Types package

Basic_DICE_Types Type Name	Corresponding element from DiceDomainModel	Kind	Values
ComputationType	DICE::DICE_UML_Extensions::DPIM::ComputationType	Enumeration	distributed, parallel, distributedParallel, microBench, sorting, grep, wordCount, collabFiltering, naiveBayes, bfs, pageRank, kMeans, connectedComponents, relQuery
TechType	DICE::DICE_UML_Extensions::DPIM::TechType	Enumeration	RDD
ProcessingType	DICE::DICE_UML_Extensions::DPIM::ProcessingType	Enumeration	synch, asynch, policy
SourceType	DICE::DICE_UML_Extensions::DPIM::SourceTypes	Enumeration	sharedStorage, dataStream
RefType	DICE::DICE_UML_Extensions::DPIM::DataSpecification	Enumeration	NoSQL, ER
RefDFType	DICE::DICE_UML_Extensions::DPIM::DataFormatType	Enumeration	RDF, JSON
ConstraintsType	DICE::DICE_UML_Extensions::DTSM::Core::ConstraintsType	Enumeration	maxIteration
ComputationalNodeType	DICE::DICE_UML_Extensions::DTSM::Core::ComputationalNodeType	Enumeration	hadoop, storm
ManagementLayerType	DICE::DICE_UML_Extensions::DTSM::Core::ManagementLayerType	Enumeration	spark
JobScheduleType	DICE::DICE_UML_Extensions::DTSM::Core::JobScheduleType	Enumeration	runtime, speculative, redundant, fair, capacity
FunctionSpecType	DICE::DICE_UML_Extensions::DTSM::Core::FunctionSpecType	Enumeration	map, reduce, combine, partition, report, collectOutput
SparkOperationKind	DICE::DICE_UML_Extensions::DTSM::Spark::SparkOperationKind	Enumeration	intersection, union, sample, count, filter, collect, map, reduce, saveAsTextFile, shuffle

B.5.2 The DICE::DICE_Library::Complex_DICE_Types package

Table 21: The DICE::DICE_Library::Complex_DICE_Types package

Complex_DICE_Types Type Name	**Corresponding element from DiceDomainModel**	**Attributes**
DiceDataVolume	DiceDomainModel::DPIM::Data	• volume: NFP_DataSize
DiceDataSpecification	DiceDomainModel::DPIM::DataSpecification	• description: String • size: NFP_DataSize • refModel: RefType • refDataFormat: RefDFType
DiceChannelSpecification	DiceDomainModel::DTSM::Core::ChannelSpecification	• rate: NFP_Frequency • size: NFP_DataSize