



# **Transformations to Analysis Models**

## **Deliverable 3.1**

<b>Deliverable:</b>	D3.1
<b>Title:</b>	Transformations to analysis models
<b>Editor(s):</b>	Abel Gómez and José-Ignacio Requeno (ZAR)
<b>Contributor(s):</b>	Danilo Ardagna (PMI), Simona Bernardi (ZAR), Marcello Bersani (PMI), Madalina Erascu (IEAT), Abel Gómez (ZAR), Christophe Joubert (PRO), Francesco Marconi (PMI), José Merseguer (ZAR), José Ignacio Requeno (ZAR) and Matteo Rossi (PMI)
<b>Reviewers:</b>	Miguel Ángel Llorente (PRO) and Youssef Ridene (NETF)
<b>Type (R/P/DEC):</b>	Report
<b>Version:</b>	1.0
<b>Date:</b>	31-July-2016
<b>Status:</b>	Final version
<b>Dissemination level:</b>	Public
<b>Download page:</b>	<a href="http://www.dice-h2020.eu/deliverables/">http://www.dice-h2020.eu/deliverables/</a>
<b>Copyright:</b>	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**

The assessment of performance and reliability properties, as well as the verification of safety properties, is a must for developing high quality software. These complex tasks need to be performed using formal models, specifically quality analysis models. This document presents the DICE transformations of UML models into quality analysis models. Therefore, it describes the model transformation techniques used to obtain simulation and verification models from the application models and their corresponding implementations and validations. Such transformations have been incorporated into the DICE-Simulation Tool [1] and the DICE-Verification Tool [2]. The work presented in this deliverable has been carried out within task T3.1 (Transformations to quality analysis models).

The DICE-Simulation Tool, DICE-Verification Tool and DICE-Profiles [3] mentioned in this document are reported in previous deliverables. All the artifacts described in this document are publicly available in the so-called DICE-Models Repository [4], DICE-Profiles Repository [5], DICE-Simulation Repository [6] and DICE-Verification Repository [7].

## Glossary

AIS	Automatic Identification System
DAM	Dependability Analysis and Modeling
DDSM	DICE Deployment Specific Model
DIA	Data-Intensive Applications
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DPIM	DICE Platform Independent Model
DTSM	DICE Technology Specific Model
EMF	Eclipse Modeling Framework
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MTM	Model To Model
M2M	Model-to-model Transformation
NFP	Non-Functional Property
NMEA	National Marine Electronics Association
OMG	Object Management Group
PNML	Petri Net Markup Language
QVT	Meta Object Facility (MOF) 2.0 Query/View/Transformation Standard
QVTc	QVT Core language
QVTo	QVT Operational Mappings language
QVTr	QVT Relations language
UML	Unified Modeling Language
VSL	Value Specification Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language



## Contents

<b>Executive summary</b>	<b>3</b>
<b>Glossary</b>	<b>4</b>
<b>Table of Contents</b>	<b>5</b>
<b>List of Figures</b>	<b>7</b>
<b>List of Tables</b>	<b>7</b>
<b>List of Listings</b>	<b>8</b>
<b>1 Introduction and Context</b>	<b>9</b>
1.1 Objectives of WP3	9
1.2 Objectives of Task T3.1	9
1.3 Objectives of this document	9
1.4 Structure of the document	10
<b>2 Requirements</b>	<b>11</b>
<b>3 Transformations Proposed for the Simulation Tool</b>	<b>13</b>
3.1 Transformations to Quality Analysis Models for the DPIM Level	14
3.1.1 DPIM DICE Stereotypes for Quantitative Analysis	14
3.1.2 Transforming DICE-profiled UML Activity Diagrams	17
3.1.3 Transforming DICE-profiled UML Sequence Diagrams	19
3.1.4 An Excerpt of the DPIM Transformations Explained	24
3.1.5 Validation of the Transformation Patterns for the DPIM Level	28
3.2 Transformations to Quality Analysis Models for the DTSM Level	30
3.2.1 Hadoop MapReduce	30
3.2.2 Storm	40
<b>4 Transformations Proposed for the Verification Tool</b>	<b>51</b>
4.1 Overview	51
4.2 DICE Models for Storm Verification	51
4.3 Transformations from DICE UML models to formal models	53
4.3.1 From DICE UML to JSON	53
4.3.2 From JSON to the Temporal Logic model	55
4.4 Validation	56
<b>5 Conclusions</b>	<b>59</b>
5.1 Further Work	60
<b>References</b>	<b>61</b>
<b>A Background</b>	<b>64</b>
A.1 Unified Modeling Language	64
A.2 Petri Net Modeling	70
A.2.1 Colored and Stochastic Petri Nets	70
A.2.2 The Petri Net Markup Language (PNML)	71
A.3 DTSM Technologies	73
A.3.1 Hadoop MapReduce Basics	73
A.3.2 Storm Basics	74

<b>B MARTE NFP Types</b>	<b>77</b>
--------------------------	-----------

## List of Figures

1	Example activity diagram (a) and corresponding Petri net (b) . . . . .	19
2	Example Sequence Diagram . . . . .	24
3	Petri net corresponding to the example Sequence Diagram (Fig. 2) . . . . .	25
4	Activity Diagram for the Parsing scenario . . . . .	29
5	Activity Diagram for the Complex Event Processing engine scenario . . . . .	29
6	Deployment Diagram . . . . .	29
7	corresponding Generalized Stochastic Petri Net model . . . . .	30
8	Example of an activity diagram for Hadoop MapReduce with DICE profile annotations .	32
9	Example of a deployment diagram for Hadoop MapReduce with DICE profile annotations	34
10	Petri net of Hadoop MapReduce . . . . .	35
11	Mapping/reducing subnets of Hadoop MapReduce . . . . .	37
12	SWN model of Hadoop MapReduce with Capacity scheduler . . . . .	38
13	Example of an activity diagram for Storm with DICE profile annotations . . . . .	41
14	Example of deployment diagram for Storm with DICE profile annotations . . . . .	44
15	Petri net for spout and bolt components in Storm . . . . .	45
16	Petri net modeling the annotation parameters of the Storm communication channel . . .	47
17	GSPN for the UML activity diagram of Storm . . . . .	48
18	<i>D-VerT</i> architecture and execution flow, highlighting the two steps of the transformation from UML Class diagram to the Temporal Logic model. . . . .	51
19	Stereotypes in DICE::Storm profile needed for verification. . . . .	52
20	Class diagram representing a simple Storm topology with “direct” subscription. Config- uration provided for each node is reported in the annotations. . . . .	52
21	Class Diagram Showing the main Java classes used to enact the transformation. . . . .	54
22	The three stages of the designed application across the transformation process: (1) DICE- profiled UML Class diagram, (2) JSON object and (3) Lisp Temporal Logic model. . . .	56
23	UML Class diagram of <i>FocusedCrawler</i> topology. . . . .	56
24	Example of a model transformation . . . . .	64
25	Metaclasses of the UML Activity Diagram . . . . .	66
26	Sample Activity Diagram . . . . .	67
27	Metaclasses of the UML Sequence Diagram . . . . .	68
28	Sample Sequence Diagram . . . . .	69
29	Metaclasses of the Petri Net Markup Language . . . . .	72
30	Storm topology with parallelism 2 and shuffle grouping (B1), and parallelism 3 and all grouping (B2) . . . . .	74
31	Multiplicity of the connections in Storm . . . . .	75
32	MARTE NFP Types (extracted from [15]) . . . . .	77
33	MARTE Measurement Units (extracted from [15]) . . . . .	78

## List of Tables

1	Transformation patterns for DICE-profiled UML Activity Diagrams . . . . .	18
2	Basic Transformation Patterns for DICE-profiled UML Sequence Diagrams . . . . .	20
3	Transformation Patterns for Combined Fragments in DICE-profiled UML Sequence Di- agrams . . . . .	22
4	DICE profile extensions for Hadoop MapReduce . . . . .	33
5	Validation of the SWN for Hadoop MapReduce . . . . .	39
6	DICE profile extensions for Storm (1) . . . . .	42
7	DICE profile extensions for Storm (2) . . . . .	43
8	Configuration parameters for the first experiment in a single workstation. . . . .	49
9	Configuration parameters for the second experiment in a cluster of two workstation. . . .	49

10	Validation of the PN when the total number of threads is greater than the number of cores	50
11	Validation of the PN when the total number of threads is less or equal than the number of cores . . . . .	50
12	Level of compliance of the current version with the initial set of requirements . . . . .	59

## List of Listings

1	Excerpt of the AD2PNML Transformation . . . . .	26
2	Excerpt of the NodeClass, representing the generic topology component . . . . .	53
3	Example JSON file describing a simple topology. . . . .	54
4	Template fragment representing the topology configuration. . . . .	55
5	JSON File produced by <i>UML2Json</i> . . . . .	56
6	Excerpt of the Lisp file produced by <i>Json2MC</i> . . . . .	58

# 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 transformations of UML models annotated with the DICE profile into suitable models that will be used by the quality analysis tools of the DICE project for performance, reliability and verification assessment.

## 1.1 Objectives of WP3

The goal of WP3 is to develop the quality analysis toolchain that will be used to guide the early design stages of the data-intensive application and guide quality evolution once operational data becomes available. In particular, the main contributions of this WP are (i) stochastic performance models and tools for simulation-based reliability and efficiency assessment, (ii) formal models and tools for formal verification of safety properties related to the sequence of events and states that the application undergoes, and (iii) numerical optimisation techniques for searching of optimal architecture designs.

The work presented in this deliverable corresponds to the task T3.1 (see below), namely, the transformations of UML models annotated with DICE profiles to quality analysis models. The transformed models are used by the DICE Simulation and Verification Tools developed in tasks T3.2 and T3.3. The tasks T3.2 and T3.3 will carry out the evaluation and verification activities. They cover the evaluation and verification activities of performance and quality annotations, data protection and privacy constraints. Finally, task T3.4 focuses on the design optimization. It assesses the impact of different architectural choices at design time, and estimates the costs associated with the usage.

## 1.2 Objectives of Task T3.1

Task T3.1 will provide the transformation of UML profiled models at DPIM and DTSM level into quality analysis models for studying a) the performance and reliability of software systems at design level; and b) the verification of safety requirements using formal verification techniques. The automatic transformation of UML models to quality analysis models requires enriching the UML diagrams with performance and safety information in order to guide the transformation process. To this end, we use the abstractions introduced by the DICE Profile (Deliverable 2.1 [3]) for describing data properties and data usage requirements among other data-related concerns. The DICE profiles provide a set of stereotypes that capture the main ideas of Big Data applications and several particular technologies. The transformations defined in the task T3.1 receive as input the design models annotated with the DICE profile and defined at DPIM and DTSM level in T2.1 and T2.2 and produce as outputs the analysis models used by the quality tools developed in the DICE project, that is, the DICE Simulation and Verification Tools.

## 1.3 Objectives of this document

This document presents the work done for the task T3.1. It explains the transformation of UML models annotated with the DICE Profile at DPIM and DTSM levels into quality analysis models for studying: 1) the performance and reliability of software systems; and 2) the verification of safety requirements. The performance models obtained by the transformation of the UML diagrams are used in the DICE Simulation Tool; and the formal models obtained for the verification of safety properties are used in the DICE Verification Tool.

In particular for the transformation to performance models, we consider UML profiled diagrams at DPIM level; and UML profiled diagrams for Hadoop MapReduce and Storm technologies at DTSM level. For the transformation to formal verification models, we consider UML profiled diagrams for the Storm technology at DTSM level. We have selected Hadoop MapReduce and Storm technologies as the initial ones in WP3 because they are well-established Big Data technologies and they are representatives for processing a set of jobs in batch or streaming mode, which are two of the main processing modes. Other technologies such as Tez or Spark are based on the previous ones or extend them in some aspects. WP3

will address some of the remaining technologies in upcoming months. This document complements the current version of deliverables D3.2 and D3.5 for the DICE Simulation and Verification tools. In particular, D3.5 has already introduced the temporal logic-based formal model of Storm topologies that is the target of the verification-oriented transformation. Hence, in the present deliverable, to avoid overlaps and unnecessary repetitions, we focus only on the mechanisms to enact the transformation from DICE profiled UML models to temporal logic models suitable for formal verification; a detailed description of the produced models can be found in [2].

## 1.4 Structure of the document

The structure of this deliverable is as follows:

- Section 1 is an executive summary.
- Section 2 summarizes the requirements that task T3.1 aims to cover.
- Section 3 summarizes the contribution of this deliverable for the transformation of the UML diagrams annotated with DICE profiles at DPIM and DTSM levels into performance models suitable for the DICE Simulation Tool.
- Section 4 summarizes the contribution of this deliverable for the transformation of the UML diagrams annotated with DICE profiles at DPIM and DTSM levels into formal models suitable for the DICE Verification Tool.
- Section 5 summarizes the goals achieved, and outlines the future work.
- Appendix summarizes the background and the standards used for the transformation process, as well as the necessary technical details of the Big Data technologies that we address.

## 2 Requirements

Deliverable D1.2 [8, 9], 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. During the progression of DICE project, the requirements and goals can be changed or adapted dynamically. For that reason, an online version of the requirement document [10] is constantly updated in order to register all the modifications and the current status.

Next, we recapitulate the requirements for Task T3.1. They will be reviewed in the Conclusions so as to evaluate the advances.

<b>ID</b>	R3.1
<b>Title</b>	M2M Transformation
<b>Priority</b>	Must have
<b>Description</b>	The TRANSFORMATION_TOOLS MUST perform a model-to-model transformation taking the input from a DPIM or DTSM DICE annotated UML model and returning a formal model (e.g. Petri net model or a temporal logic model).
<b>ID</b>	R3.2
<b>Title</b>	Taking into account relevant annotations
<b>Priority</b>	Must have
<b>Description</b>	The TRANSFORMATION_TOOLS MUST take into account the relevant annotations in the DICE profile (properties, constraints and metrics) whether related to performance, reliability, safety, privacy, and transform them into the corresponding artifact in the form.
<b>ID</b>	R3.3
<b>Title</b>	Transformation rules
<b>Priority</b>	Could have
<b>Description</b>	The TRANSFORMATION_TOOLS MAY be able to extract, interpret and apply the transformation rules from an external source.
<b>ID</b>	R3.6
<b>Title</b>	Transparency of underlying tools
<b>Priority</b>	Must have
<b>Description</b>	The TRANSFORMATION_TOOLS and SIMULATION_TOOLS MUST be transparent to users. From their point of view the user is analyzing metrics from and making simulations over an enriched UML Model.

<b>ID</b>	R3.7
<b>Title</b>	Generation of traces from the system model
<b>Priority</b>	Must have
<b>Description</b>	The VERIFICATION_TOOLS MUST be able, from the UML DICE model a system, to show possible execution traces of the system, with its corresponding time stamps. This sequence SHOULD be used by the QA_ENGINEER to determine whether the system model captures the behavior of the application or not, for model validation purposes.
<b>ID</b>	R3.12
<b>Title</b>	Modelling abstraction level
<b>Priority</b>	Must have
<b>Description</b>	Depending on the abstraction level of the UML models (detail of the information gathered, e.g., about components, algorithms or any kind of elements of the system we are reasoning about), the TRANSFORMATION_TOOLS will create the formal model accordingly, i.e., at that same level that the original UML model.
<b>ID</b>	R3.13
<b>Title</b>	White/black box transparency
<b>Priority</b>	Must have
<b>Description</b>	For the TRANSFORMATION_TOOLS and the SIMULATION_TOOLS there will be no difference between white box and black box model elements.
<b>ID</b>	R3.15
<b>Title</b>	Verification of temporal safety/privacy properties
<b>Priority</b>	Must have
<b>Description</b>	Taking the DICE annotated UML model (which must include the property to be verified) as an input, the VERIFICATION_TOOLS MUST be able to answer questions related to whether the specified property holds for the modeled system or not.
<b>ID</b>	R3IDE.3
<b>Title</b>	Usability
<b>Priority</b>	Could have
<b>Description</b>	The TRANSFORMATION_TOOLS and SIMULATION_TOOLS MAY follow some usability, ergonomics or accessibility standard such as ISO/TR 16982:2002, ISO 9241, WAI W3C or similar.



### 3 Transformations Proposed for the Simulation Tool

One of the tools within the DICE framework is the so-called *Simulation Tool*. It allows evaluating quality properties of data-intensive applications, in particular efficiency and reliability metrics. The DICE Simulation Tool considers annotated UML models at the DPIM and DTSM level, and returns information about the prediction of a metric value in the environment being studied. Models at the DPIM layer specify the fundamental architecture elements that constitute a data-intensive application. For example, they include the definition of the 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.). A DPIM can be enriched with technological information and then transformed into a model at DTSM level. It defines a more precise and concrete view of the execution platform of the system. The next step consists of filling the gap between the DPIM and DTSM and the formal models needed for performance assessment.

The objective of this section is the definition of an automatic transformation toolchain within the DICE framework that receives a UML model annotated with the DICE profile at DPIM or DTSM level and finally obtains a formal model suitable for performance and reliability analysis in the DICE Simulation Tool. A plain UML model must be annotated with extra information that drives the transformation process so that the UML diagram can be transformed into adequate models for the simulation and prediction. To this end, we use a set of DICE profiles presented in the Deliverable 2.1 [3]. The DICE profiles define UML stereotypes that capture the essential attributes for the data-intensive applications (DIA) and the underlying Big Data technology at DPIM and DTSM level. The stereotypes of the DICE profiles can be seen as templates that are instantiated with the parameters of the system in the performance model. The designer only has to tune some particular values for the system being modeled.

In particular, the methodology that we use in this document for obtaining a formal model for performance assessment from the UML diagram is divided in four steps: (i) the identification of the main application and technological concepts that are relevant for the system that is being modeled and the correspondence with the stereotypes and annotations of the DICE Profile, (ii) the theoretical definition of the transformation from an annotated UML model with the DICE profile into a formal model for performance assessment according to the current aspects annotated in a UML diagram, (iii) the validation of the transformation process by comparing the results returned with the simulation tool for the performance model obtained by the transformation toolchain and the results returned by the real execution of the application on a controlled environment, and (iv) the implementation of the automatic transformation of UML diagrams to performance models. This section covers the points (i)-(iv) for the DPIM and (i)-(iii) for the DTSM. The point (iv) for the DTSM is currently under development.

The transformation of the Petri net represented in the PNML format into the format of a specific Petri net tool is accomplished using Model-to-Text (M2T) transformations. To execute the M2T transformations we have selected Acceleo [11]. Starting from version 3, Acceleo transformations are specified using the MOFM2T standard language [12], proposed by the OMG too. In this sense, we have selected Acceleo to make all our toolchain compliant with the OMG standards, from the definition of the initial (profiled) UML models to the 3rd party analysis tools (which use a proprietary format). In our case, the Petri net tool that receives the resulting Petri net at the end of the model transformation toolchain is GreatSPN [13].

The structure of this section is divided in two parts: the first one is devoted to the actual transformations of the UML models annotated with the DICE profile at the DPIM level; and the second one is devoted to actual the transformations of the UML models annotated with the DICE profile at the DTSM level.

In the first part of this section, we focus on the transformations at the DPIM level. DPIM models are independent of any platform and technology, and as such, any behavior that needs to be analysed must be modeled explicitly. Such modeling is done using two UML behavioral models, namely, *UML Activity Diagrams* and *UML Sequence Diagrams*, that are complemented using the stereotypes of the aforementioned DICE profile. Consequently, in this subsection, (i) we describe the stereotypes provided by the DPIM DICE profile that are of interest for the transformations to analysis models; (ii) we describe

the transformation patterns used to transform *UML Activity Diagrams*; (iii) we describe the transformation patterns used to transform *UML Sequence Diagrams*; (iv) we explain an excerpt of the actual QVT transformation that perform the automatic translation from UML to analysis models; and (v) we present the experiments that validate the patterns presented.

In the second part of this section, we focus on the transformations of *UML Activity Diagrams* and *UML Deployment Diagrams* at the DTSM level for two of the main technologies for DIA applications: Apache Hadoop MapReduce and Apache Storm. Each technology has a dedicated subsection. On the one hand, Hadoop MapReduce is a distributed cluster-based generalization of the map/reduce functions from functional programming paradigm that processes the tasks in batch mode according to some scheduling and fault-tolerance policies. On the other side, Storm is a distributed computation framework for real-time processing of data streams in a directed acyclic graph (DAG) topology (i.e., a Storm application). These technologies cover the main families of computational process, both batch and streaming approaches. Other technologies such as Spark and Tez are variants or refinements of the technologies studied here and the methodology presented in this section for obtaining performance models from annotated UML diagrams can be applied to those technologies too.

### 3.1 Transformations to Quality Analysis Models for the DPIM Level

Performance evaluation is traditionally carried out using scenarios, i.e., typical system paths of usage that specify the system behavior of an application. With UML, we can specify a scenario by using behavioral diagrams, such as the previously presented activity or sequence ones. However to perform a quantitative analysis of such scenarios, UML models need to be complemented with some quantitative data. Here is where the DICE profile gets into the action.

At the DPIM layer, the DICE profile provides Software Architects with a set of core concepts to specify the fundamental architecture elements that constitute a data-intensive application, together with the high level topology of the application and its QoS requirements. 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.).

In this Section, first we describe which DICE stereotypes are relevant for performance evaluation; second, we describe the model transformations that we have developed in the DICE Simulation Tool to analyse DIA models at the DPIM layer; third, we explain an excerpt of the DPIM QVT to transformations to illustrate what the transformation rules look like; and fourth, we describe how we have validated the proposed transformation patterns in one of the case studies of DICE: POSIDONIA Operations [14].

#### 3.1.1 DPIM DICE Stereotypes for Quantitative Analysis

Activity and Sequence diagrams are the two behavioral diagrams of UML that DICE considers for performance evaluation. Both kinds of diagrams need to be annotated with stereotypes from the DICE profile in order to be simulated. These annotations guide the transformation from the UML domain to the Petri net domain, and more specifically, to the PNML abstract syntax presented in Section A.2.2. Several DICE stereotypes can be applied to elements in both kinds of diagrams to express the same. For that reason, next we summarize the main stereotypes that may be used in such diagrams to specify the NFP that are relevant to perform a simulation.

**DICE::DICE\_UML\_Extensions::DPIM::DpimScenario** — A Scenario (*DpimScenario*) captures system-level behavior and attaches allocations and resource usages to it. It is composed of suboperations called *Steps* (GaStep).

- In DICE, applied to:

**UML::Activities::Activity** (as a specialization of **UML::Classifiers::NamedElement**) in *Activity Diagrams*.

**UML::Interactions::Interaction** (as a specialization of **UML::Classifiers::NamedElement**) in *Sequence Diagrams*.

- Tagged values of interest:

**throughput** : MARTE\_Library::Basic\_NFP\_Types::NFP\_Frequency [\*]

**respT** (response time): MARTE\_Library::Basic\_NFP\_Types::NFP\_Duration [\*]

**utilization** : MARTE\_Library::Basic\_NFP\_Types::NFP\_Real [\*]

#### Example:

The following expression applied to a *DpimScenario* specifies that the simulation should calculate (calc) the mean *response time* (respT) of the scenario, in seconds. That response time should be associated to the \$rt variable.

```
respT = (expr = $rt, unit = s, statQ = mean, source = calc)
```

**MARTE::MARTE\_AnalysisModel::GQAM::GaAnalysisContext** — For a given analysis, the context identifies the model elements (diagrams) of interest and specifies global parameters of the analysis.

- In DICE, applied to:

**UML::Activities::Activity** (as a specialization of **UML::Classifiers::NamedElement**).

- Tagged values of interest:

**contextParams** : NFP\_String [\*]

Strings giving a set of annotation variables defining global properties of this analysis context. Each string should conform to the concrete syntax for variable calls or declarations as defined in B.3.3.12 of the MARTE standard.

Variable names must match the following structure:

```
identifier ::= ("$(letter|"_") (letter | digit | "_")*)
```

#### Example:

The following expression specifies that \$rt is an output variable, and \$njobs, \$p1 and \$t1 are input variables (if unspecified – e.g., \$t1 – a variable is considered as input variable). Additionally, the default value for \$njobs is 5.

```
contextParams = [ out$rt, in$njobs = 5, in$p1, $t1 ]
```

**MARTE::MARTE\_AnalysisModel::GQAM::GaWorkloadEvent** — A stream of events that initiate system-level behavior. It may be generated in different ways: by a stated arrival process, by an arrival-generating mechanism modeled by a workload generator class, by a timed event and from a trace.

- In DICE, applied to:

**UML::Activities::InitialNode** in an Activity Diagram.

**UML::Interactions::Message** (only the first message) in Sequence Diagram.

- Tagged values of interest:

**pattern** : MARTE::MARTE\_Library::BasicNFP\_Types::ArrivalPattern [0..1]

Pattern of arrival events. The pattern can be:

**closed** — It describes a workload characterized by a fixed number of active or potential users or jobs that cycle between executing the scenario. This pattern makes the transformation produce a closed Petri net that is analysed in steady state. The following attributes may be defined:

**population** : NFP\_Integer [0..1]

Size of the workload. This property is required for the automatic analysis of the net, and denotes the initial marking of the place corresponding to this *InitialNode*.

**extDelay** : NFP\_Duration [0..1]

The delay between the end of one response and the start of the next for each member of the population of system users.

**open** — This pattern produces an open Petri net for transient analysis with initial and final transitions that produce and consume tokens. The following mutually exclusive attributes may be defined:

**interArrivalTime** : NFP\_Duration [0..1]

The time between successive arrivals. For a Poisson process this is exponentially distributed with  $mean = 1/rate$ .

**arrivalRate** : NFP\_Frequency [0..1]

The average rate of arrivals.

### Example:

A valid GaWorkloadEvent denoting an initial marking of \$njobs is typically declared as follows:

```
pattern = (closed = (population = (expr = $njobs)))
```

**MARTE::MARTE\_AnalysisModel::GQAM::GaStep** (extends MARTE::MARTE\_AnalysisModel::GQAM::GaScenario) — A *GaStep* is a part of a *Scenario*, defined in sequence with other actions.

- In DICE, applied to:

**UML::Action::Action** in an Activity Diagram using the throughput or utilization tagged values to calculate the metric on this *Action*.

**UML::Action::Action** in an Activity Diagram using the hostDemand tagged value to specify its mean execution time.

**UML::Activities::ControlFlow** (as a specialization of UML::Classifiers::NamedElement) in an Activity Diagram using the prob tagged value to specify the probability of the execution path after a *ChoiceNode*.

**UML::Interactions::ExecutionSpecification** in an Sequence Diagram using the hostDemand tagged value to specify its mean execution time.

**UML::Interactions::Message** in an Sequence Diagram using the hostDemand tagged value to specify its mean communication time.

**UML::Interactions::InteractionOperand** in an Sequence Diagram using the prob tagged value to specify the probability of executing that execution path.

- Tagged values of interest:

**hostDemand** : MARTE\_Library::Basic\_NFP\_Types::NFP\_Duration [\*]

The cpu demand in units of operations, if all *Steps* are on the same host.

**prob** : MARTE\_Library::Basic\_NFP\_Types::NFP\_Real [0..1] = 1

The probability of the step to be executed (for a conditional execution).

### Examples:

The following hostDemand declarations can be applied to an *Action* to specify that the mean execution time is 2 seconds.

```
hostDemand = (value = 2, unit = s)
hostDemand = (value = 2, unit = s, statQ = mean)
```

The following `hostDemand` declaration can be applied to an *Action* to specify that the mean (if unspecified, the `statQ` is assumed to be the mean) execution time is `$t1` seconds.

```
hostDemand = (expr = $t1, unit = s)
```

The following `hostDemand` declaration can be applied to a *ControlFlow* to specify that the probability of this alternative path is `$p1`.

```
prob = (expr = $p1)
```

**MARTE::MARTE\_AnalysisModel::PAM::PaRunTInstance** — A *PaRunTInstance* is a stereotype for a swimlane or lifeline that indicates a run-time instance of a process resource and its properties.

- In DICE, applied to:  
**UML::Interactions::Lifeline** in an Sequence Diagram using the `poolSize` tagged value to specify the number of threads of the process.
- Tagged values of interest:  
**poolSize** : MARTE\_Library::Basic\_NFP\_Types::NFP\_Integer [0..1]  
The number of threads for the process.

**Example:**

The following `poolSize` declarations can be applied to a *Lifeline* to specify the number of threads that the lifeline represents.

```
poolSize = (value = 2)
```

### 3.1.2 Transforming DICE-profiled UML Activity Diagrams

The *Activity Diagram To PNML Transformation* (AD2PNML) is the transformation in charge of producing analysable Petri nets out of DICE-profiled UML Activity Diagrams. To implement this transformation, a set of transformation patterns have been identified between the two domains (UML and PNML). Later on, following the MDE paradigm, these transformation patterns have been first mapped to the concepts of the domain metamodels, and second translated to a set of transformation rules written in QVTo.

Next, we first describe the transformation patterns using a graphical syntax; and second, we present a small illustrative example that demonstrates what the transformation produces when applied to a complete *Activity Diagram*.

#### Transformation Patterns

Table 1 summarizes the main transformation patterns that we have identified between the DICE-profiled activity diagrams and the Petri net domain. The first column indicates to which UML elements the pattern applies and which stereotypes may be applied to them. The second column shows the pattern, in the UML domain, that will be checked against the source candidate model to find possible matches. The third column shows the Petri net fragment that will be generated from the elements matched by the pattern in the second column. In Petri net fragments,  $M(x)$  represents the initial marking for element  $x$ ;  $r(y)$  represents the firing rate of transition  $y$ , and  $w(z)$  represent the weight (i.e., firing probability) of transition  $z$ . Elements in black represent the elements that are actually transformed by the pattern, and elements in gray are only shown to provide additional information about the matching context (e.g., which element may precede or follow the matching pattern).

Table 1: Transformation patterns for DICE-profiled UML Activity Diagrams

UML ELEMENT(S)	UML PATTERN	PETRI NET PATTERN
(1) <i>InitialNode</i> stereotyped as «GaWorkloadEvent» (closed pattern)  <i>ActivityFinalNode</i>	<p>«GaWorkloadEvent» closed=(population=\$pop, extDelay=\$delay)</p>	<p><math>M(p_1)=\\$pop</math> <math>r(t_1)=1/\\$delay</math></p>
(2) <i>InitialNode</i> stereotyped as «GaWorkloadEvent» (open pattern)  <i>ActivityFinalNode</i>	<p>«GaWorkloadEvent» open=(arrivalRate=\$rate)</p>	<p><math>r(t_1)=\\$rate</math></p>
(3) <i>FlowFinalNode</i>		
(4) <i>OpaqueAction</i> stereotyped as «GaStep»	<p>«GaStep» hostDemand=(expr=\$time, unit=s, source=est, statQ=mean)</p>	<p><math>r(t_A)=1/\\$time</math></p>
(5) <i>DecisionNode</i>		
(6) <i>MergeNode</i>		
(7) <i>ForkNode</i>		
(8) <i>JoinNode</i>		
(9) <i>ActivityPartition</i> linked via its <i>represents</i> property to an element stereotyped as «Resource»		<p><math>M(p_R)=\\$size</math></p>
(10) <i>ControlFlow</i> (general case)		
(11) <i>ControlFlow</i> stereotyped as «GaStep» (departing form a <i>DecisionNode</i> )	<p>«GaStep» prob=(expr=\$p1 unit=s, source=est, statQ=mean)</p>	<p><math>w(t_1)=\\$p1</math></p>
(12) <i>ControlFlow</i> (arriving to a <i>JoinNode</i> )		



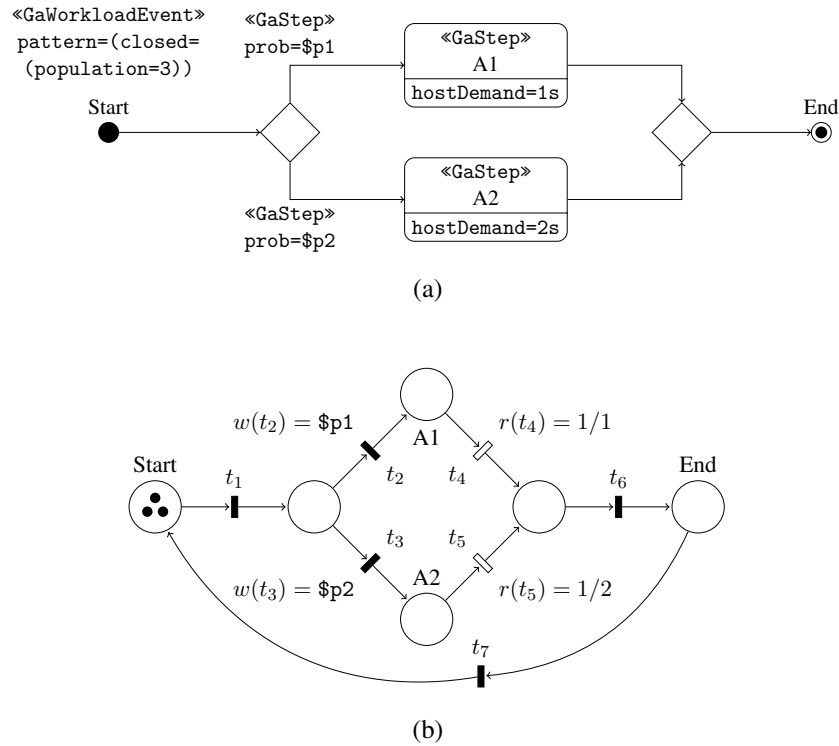


Figure 1: Example activity diagram (a) and corresponding Petri net (b)

### An illustrative example

Figure 1a shows a simple UML activity diagram consisting of a unique initial node (*Start*), a unique final node (*End*) and two alternative actions (*A1* and *A2*). Observe that the two *GaWorkloadEvent* and *GaStep* stereotypes have been applied. Such stereotypes, defined in the MARTE profile [15], are imported in the DICE profile to specify the performance input parameters. As previously explained, *GaWorkloadEvent* is applied to the initial node to specify a closed workload with an initial population of 3 jobs, and *GaStep* is applied to the transitions outgoing the decision node to indicate the probability of execution of the actions *A1* o *A2* i.e.,  $\$p1$  and  $\$p2$ , respectively. Such probabilities are specified as variables that will be set to actual values when configuring the simulation experiments. Finally, the *GaStep* stereotype is applied to the actions *A1* and *A2* to indicate the CPU host demand required for their execution, i.e., 1s in case of action *A1* and 2s in case of action *A2*.

Figure 1b shows the graphical representation of the GSPN model derived from the M2M transformation and that can be used to evaluate the performance of the system specified by the Activity Diagram in Figure 1a: circles correspond to *places*, black bars represent *immediate transitions* and white bars represent *timed transitions*. The initial marking of the *Start* place consists of three tokens (as indicated by the initial population annotated in the UML model); immediate transitions  $t_2$  and  $t_3$  are characterized by weights  $\$p1$  and  $\$p2$ , respectively; and the timed transitions  $t_4$  and  $t_5$  are characterized by the firing rates (inverse of the mean firing times annotated in the host demand tagged values).

### 3.1.3 Transforming DICE-profiled UML Sequence Diagrams

The *Sequence Diagram To PNML Transformation* (SD2PNML) is the transformation in charge of transforming DICE-profiled UML Sequence Diagrams to analysable Petri nets. Similarly to the AD2PNML transformation, a set of transformation patterns have been identified between the two domains, which in turn, have been translated to a set of transformation rules written in QVTo.

Next, we first describe the transformation patterns using a graphical syntax; and second, we present a small example demonstrating the transformation.

## Transformation Patterns

Table 2 summarizes the main transformation patterns that we have identified between the DICE-profiled sequence diagrams and the Petri net domains. The first column indicates to which UML elements the pattern applies and which stereotypes may be applied to them. The second column shows the pattern, in the UML domain, that will be checked against the source candidate model to find possible matches. The third column shows the Petri net fragment that will be generated from the elements matched by the pattern in the second column. As in table 1, elements in black represent the elements that are actually transformed by the pattern and elements in gray are only shown to provide additional information about the matching context. In addition to these conventions, since *OccurrenceSpecifications* are not represented using any graphical primitive, a solid black circle drawn on top of a lifeline indicates that an instance of an *OccurrenceSpecification* is of interest for the transformation pattern. These black circles can be drawn at the top or the bottom of an *ExecutionSpecification* to represent *ExecutionOccurrenceSpecifications*; and the start or the end of a *Message* to represent a *MessageOccurrenceSpecification*.

Table 2: Basic Transformation Patterns for DICE-profiled UML Sequence Diagrams

UML ELEMENT(S)	UML PATTERN	PETRI NET PATTERN
<p>(1) Lifeline sending the first message, which is stereotyped as «GaWorkloadEvent» (open pattern)</p>		
<p>(2) Lifeline sending the first message, which is stereotyped as «GaWorkloadEvent» (closed pattern)</p>		

continued ...

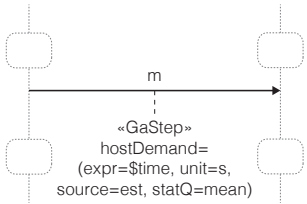
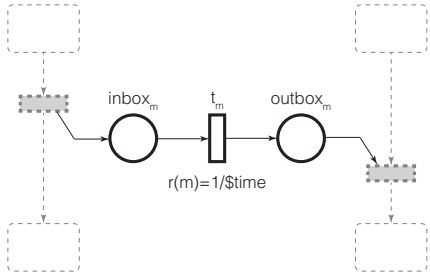


...continued

UML ELEMENT(S)	UML PATTERN	PETRI NET PATTERN
(3) All other Lifelines stereotyped as «PaRunTInstance»		
(4) Execution-Occurrence-Specification		
(5) ExecutionSpecification stereotyped as «GaStep»		
(6) MessageOccurrence-Specification (send event)		
(7) MessageOccurrence-Specification (receive event)		

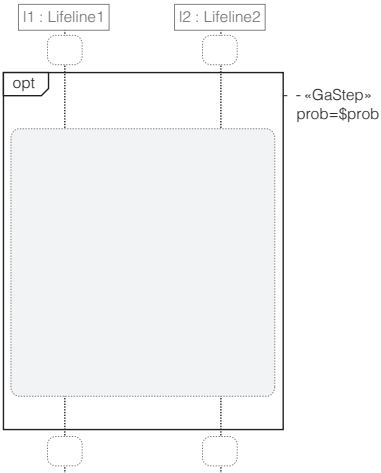
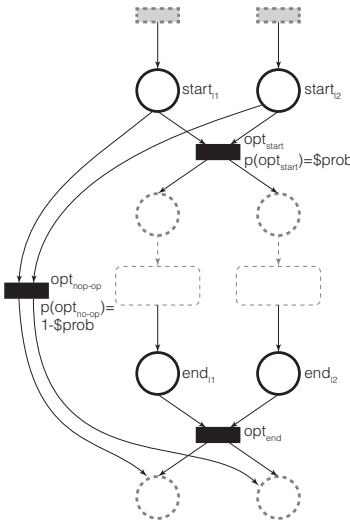
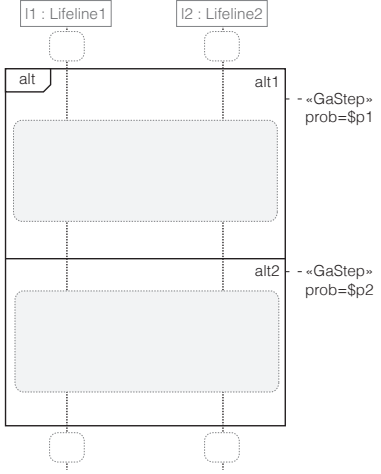
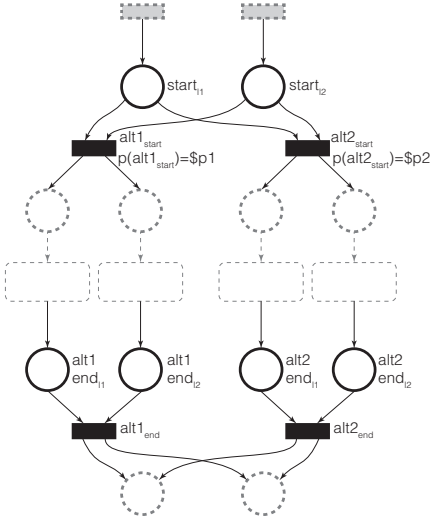
continued ...

...continued

UML ELEMENT(S)	UML PATTERN	PETRI NET PATTERN
(8) <i>Message</i> stereotyped as «GaStep»		

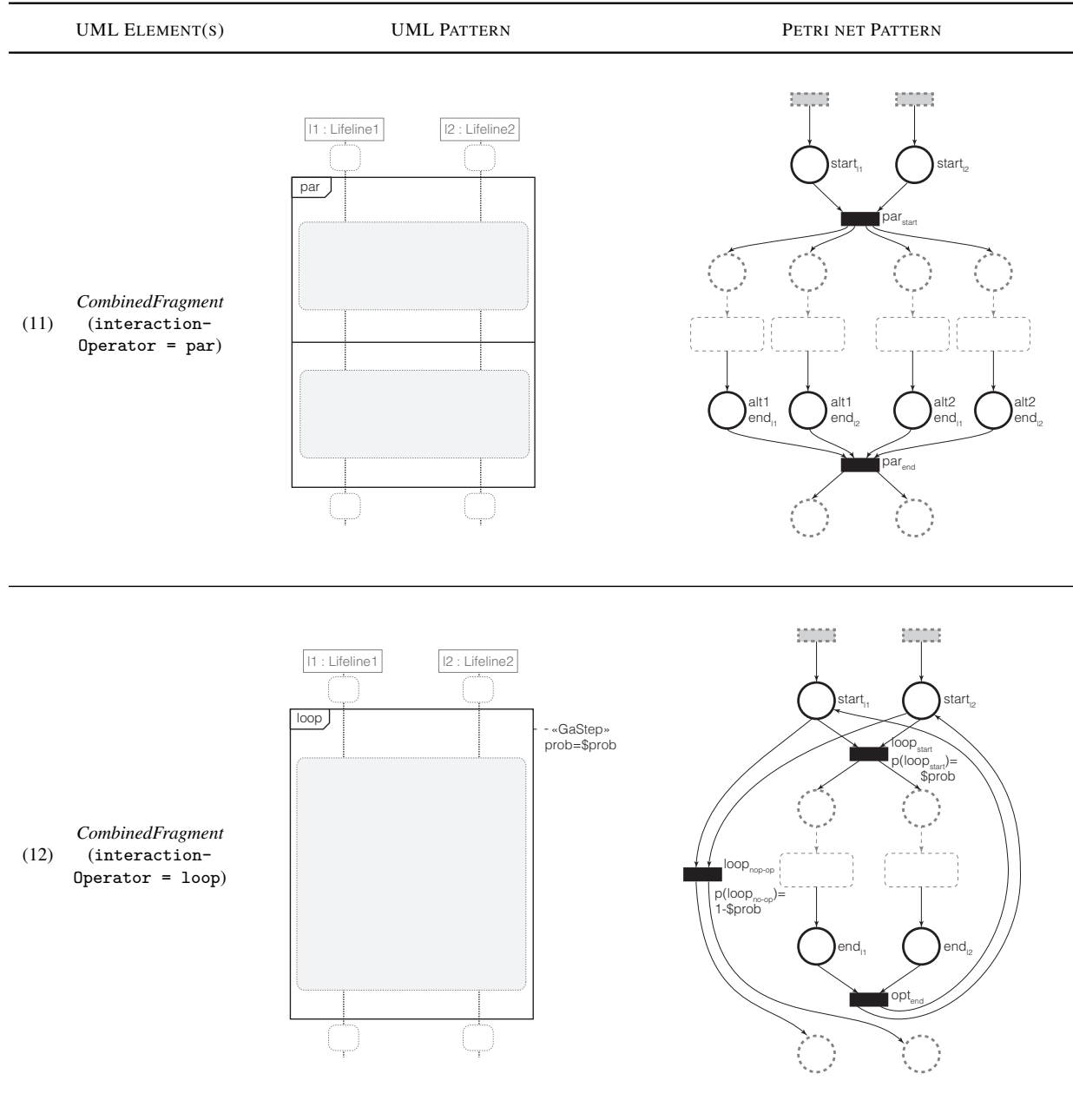
*CombinedFragments* produce more complex patterns. For the sake of simplicity, Table 3 presents the patterns to transform the most used *CombinedFragment* types. To present these patterns, we have limited the number of lifelines covered by the *CombinedFragments* to two (i.e., *l1* and *l2*). Nevertheless, observing these patterns carefully, it can be observed that they can be easily extended to an arbitrary number of *Lifelines* or *InteractionOperands*.

Table 3: Transformation Patterns for Combined Fragments in DICE-profiled UML Sequence Diagrams

UML ELEMENT(S)	UML PATTERN	PETRI NET PATTERN
(9) <i>CombinedFragment</i> (interaction-Operator = opt)		
(10) <i>CombinedFragment</i> (interaction-Operator = alt)		

continued ...

...continued



## An illustrative example

Figure 2 on the following page shows an example *Sequence Diagram* with some DICE stereotypes applied. Specifically, the diagram contains three lifelines: the first one (*a*) represents the *Actor* that generates the workload of the system; the second one (*l1*) represents an instance of the *Lifeline1* classifier with, a *poolSize* of *\$size1*; and the third one (*l2*) represents an instance of the *Lifeline2* classifier with, a *poolSize* of *\$size2*.

The workload of the system is specified in the first message (*m1*) that *a* sends to *l1* by using the *GaWorkloadEvent* stereotype. As it can be observed, the workload follows an open pattern, which specifies an arrival rate of *\$rate*. When *l1* receives the *m1* message, it starts an *ExecutionSpecification* that, optionally, may send a message *m2* to *l2*, which in turn, will start an *ExecutionSpecification* that will take *\$time* seconds (on average) to be executed. At the end of that execution, *l2* will send a reply message to *l1* and *l1* will send a reply message to *a*. In the case of *l1* not sending a message to *l2*, *l1* will send a reply message to *a* directly.

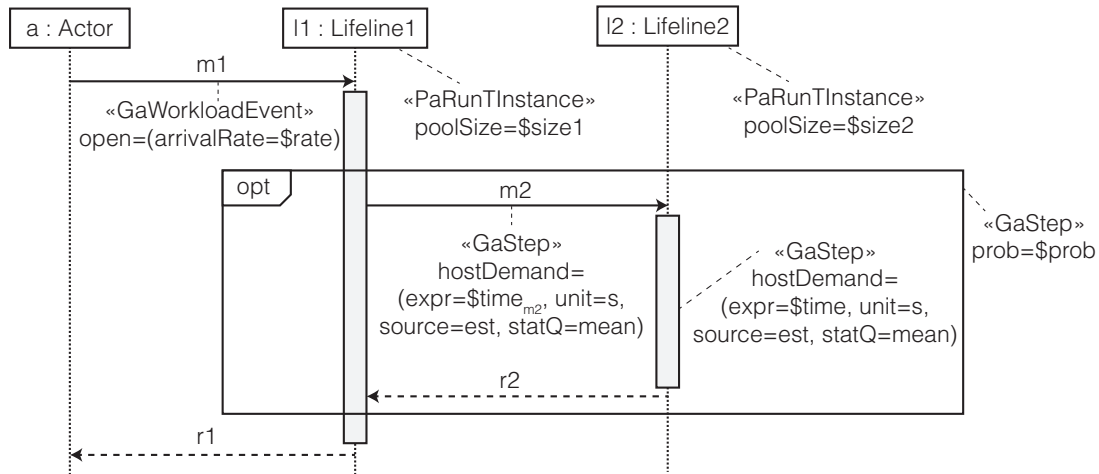


Figure 2: Example Sequence Diagram

Figure 3 shows the Petri net that corresponds to the *Sequence Diagram* depicted in Figure 2. That net has been produced by applying the transformation patterns previously explained to the initial DICE-profiled UML model. In the figure, places and transitions have been arranged vertically in three imaginary lines that represent each one of the execution paths of original lifelines. Thus, places and transitions in the leftmost vertical line correspond to UML elements placed in the lifeline of *a*; places and transitions in the middle vertical line correspond to UML elements placed in the lifeline of *l1*; and places and transitions in the rightmost vertical line correspond to UML elements placed in the lifeline of *l2*.

As it can be observed, the leftmost execution path is left open, as indicated by pattern (1) of Table 2. The middle and the rightmost execution paths converge in the *common* transition as indicated by pattern number (3). Places and transitions that communicate the different execution paths (e.g.,  $inbox_{m1}$ ,  $m1outbox_{m1}$ ,  $inbox_{m2}$ ,  $m2$ ,  $outbox_{m2}$ ,  $inbox_{r1}$ ,  $r1$ ,  $outbox_{r1}$ ,  $inbox_{r2}$ ,  $r2$  and  $outbox_{r2}$ ) represent the messages between lifelines. The optional execution path has been transformed according to pattern (9). As it can be observed, the communication between *l1* and *l2* can be avoided by following the path that passes through the transition  $opt_{no-op}$ . Finally, the *ExecutionSpecification* of *l2* can be found in the timed transition ( $t_{exe}$ ) placed in the rightmost execution path. As it can be observed,  $t_{exe}$  has a firing rate ( $r(t_{exe})$ ) of  $1/\$time$  as specified in the source UML diagram.

### 3.1.4 An Excerpt of the DPIM Transformations Explained

Listing 1 on page 26 reproduces a simplified version of the QVTo rules that are in charge of transforming pattern number (4) of Table 1 – i.e., an *OpaqueAction* of an Activity Diagram stereotyped as a «*GaStep*» – into a Petri net fragment formed by a Place, an Arc, and a Transition. The complete DPIM transformations (AD2PNML and SD2PNML) can be found in the Companion Document [16] and a complete reference of the QVTo language specification can be found in Chapter 8 of the QVT standard [17].

As it can be observed in line 1 of the listing, the transformation declares four arguments, two of them are the inputs and the other two are the outputs. The two input arguments are: (i) *ad*, a DICE-profiled UML model; and (ii) *vars*, a set of  $\langle \$var, value \rangle$  pairs used to valuate the VSL expressions. On the other hand, the remaining (output) arguments are (iii) *res*, the PNML result model; and (iv) *traces*, a set of traceability links that relate the element of the input UML model with the elements of the output PNML model.

Next, the *main()* method – the entry point for the transformation – is declared. It invokes the *mapping* (e.g., rule) *activityNode2subNet* for each *ActivityNode* contained in the scenario being transformed. When the actual type of the *ActivityNode* being transformed is *OpaqueAction*, the *activityNode2subNet* *mapping* delegates its execution to *basicActivityNode2subNet* since it is the only *mapping* whose type matches the type of the element being transformed. The *basicActivity-*

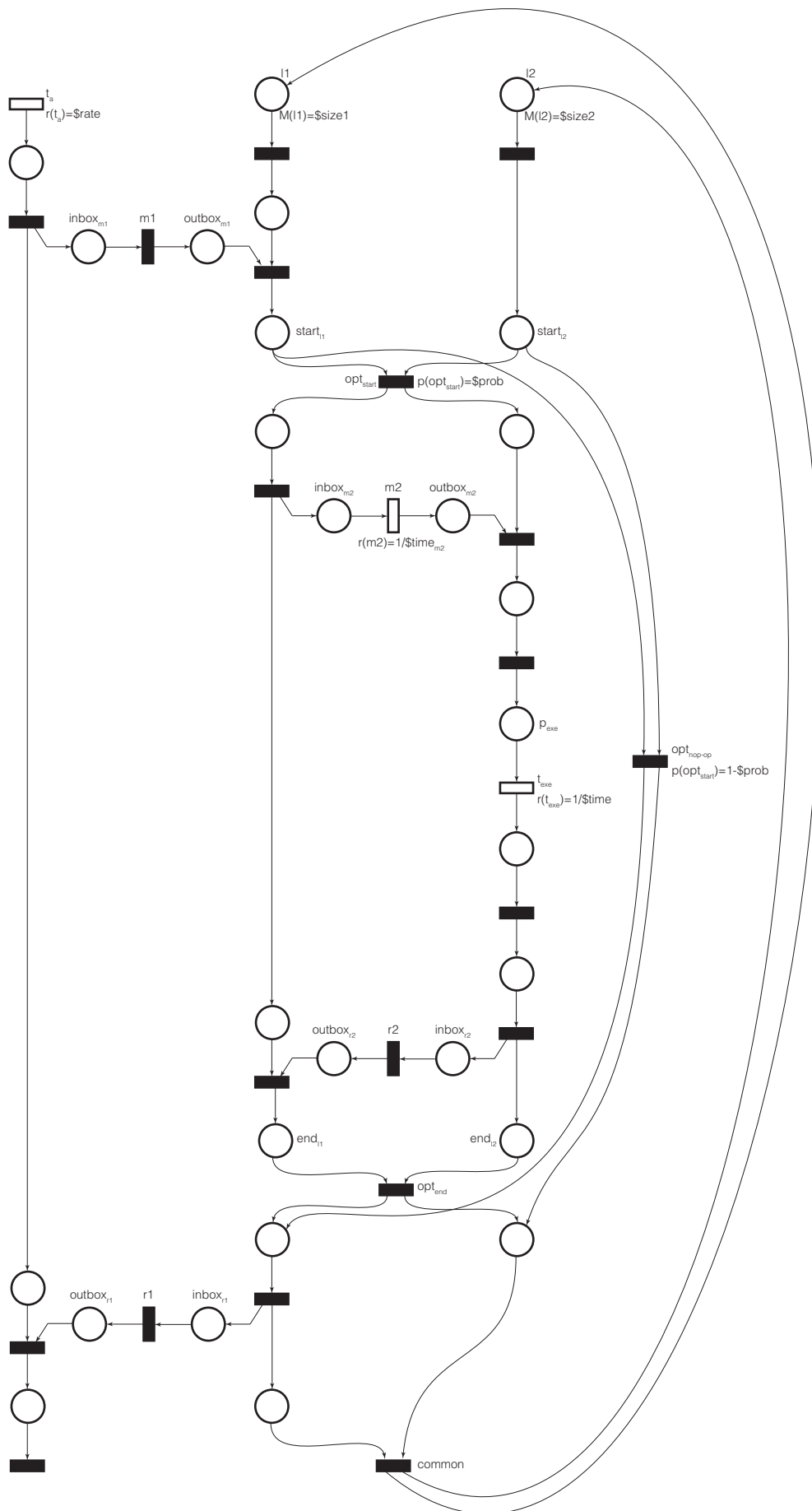


Figure 3: Petri net corresponding to the example Sequence Diagram (Fig. 2)

Node2subNet *mapping* is a simple rule that, by calling the *activityNode2place* and *activityNode2transition mappings*, creates a *Place* and a *Transition*. Next, it creates the arc that links the *Place* with the *Transition* by calling the *mapping* *arc(...)*. Finally, the mapping creates two trace links by calling the *trace(...)* mapping: the first one links the *ActivityNode* to the newly created *Place*; and the second one links the *ActivityNode* to the newly created *Transition*.

Regarding the *activityNode2transition* mapping, it is noteworthy to highlight its last line: this line invokes the *opaqueActionHostDemand2toolInfo mapping* in the case of the element being transformed is an *OpaqueAction*. Specifically, that rule (*opaqueActionHostDemand2toolInfo*) will check whether the *OpaqueAction* has a *hostDemand* NFP defined or not. In the case of *OpaqueAction* having the *GaStep* stereotype applied, if it contains a *hostDemand* tagged value, the rule will generate the metadata (in the form of a PNML *ToolInfo* element) specifying the transition processing rate according to the transformation patterns.

Listing 1: Excerpt of the AD2PNML Transformation

```

1 transformation ad2pnml(in ad : UML, in vars : TYPES, out res : PNML, out traces :
  TRACE);
2
3 main() {
4 // [...] Content removed for clarity purposes
5 ad.scenario().node[UML::ActivityNode] -> map activityNode2subNet();
6 // [...] Content removed for clarity purposes
7 }
8
9 mapping UML::ActivityNode::activityNode2subNet() disjuncts
10 UML::InitialNode::initialNode2subNet,
11 UML::DecisionNode::decisionActivityNode2subNet,
12 UML::JoinNode::joinActivityNode2subNet,
13 UML::ActivityNode::basicActivityNode2subNet {};
14
15 /**
16 Transform a generic ActivityNode into a simple [place]->[transition] subnet
17 */
18 mapping UML::ActivityNode::basicActivityNode2subNet() {
19 var place := self.map activityNode2place();
20 var transition := self.map activityNode2transition();
21 var arc := map arc(place, transition);
22 // Add tracing information
23 self.map trace(place, "basicActivityNode2place");
24 self.map trace(transition, "basicActivityNode2transition");
25 }
26
27 /**
28 Transform a generic ActivityNode into a Place
29 */
30 mapping UML::ActivityNode::activityNode2place() : PNML::Place {
31 containerPage := resolveoneIn(UML::NamedElement::model2page);
32 id := createRandomUniqueId();
33 if (self.name.oclIsUndefined().not()) {
34 name := object PNML::Name {
35 text := self.name;
36 };
37 };
38 }
39
40 /**
41 Transform a generic ActivityNode into a Transition and
42 creates any additional ToolInfo depending on the ActivityNode
43 subtype (e.g., OpaqueActions with hostDemand may create

```

```

44     exponential transitions)
45 */
46 mapping UML::ActivityNode::activityNode2transition() : PNML::Transition {
47     containerPage := resolveOneIn(UML::NamedElement::model2page);
48     id := createRandomUniqueId();
49     if (self.name.ocllsUndefined().not()) {
50         name := object PNML::Name {
51             text := self.name;
52         };
53     };
54     toolspecifics += self[OpaqueAction].map opaqueActionHostDemand2toolInfo();
55 }
56
57 /**
58  Transformas an OpaqueAction with a hostDemand annotation to a ToolInfo element
59 */
60 mapping UML::OpaqueAction::opaqueActionHostDemand2toolInfo() : List ( PNML::ToolInfo )
61 when {
62     self.getGaStep_hostDemand().ocllsUndefined().not();
63 }{
64     var hostDemand := self.getGaStep_hostDemand();
65     result += expTransitionToolInfo( 1 / hostDemand.value());
66     result += infServerTransitionToolInfo();
67 }
68
69 /**
70  Creates the ToolInfo that identifies an exponential timed transition,
71  i.e., CONST::TransitionKind::Exponential
72 */
73 helper expTransitionToolInfo(rate : Real) : PNML::ToolInfo {
74     return object PNML::ToolInfo {
75         tool := CONST::ToolInfoConstants::toolName.toString();
76         version := CONST::ToolInfoConstants::toolVersion.toString();
77         toolInfoGrammarURI := CONST::TransitionKind::Exponential.toString().createURI();
78         formattedXMLBuffer := ("<value grammar=\"" +
79             CONST::TransitionKind::Exponential.toString() + "\">" + rate.toString() +
80             "</value>").createLongString();
81     };
82 }
83
84 /**
85  Creates the ToolInfo that identifies an InfiniteServer timed transition,
86  i.e., CONST::ServerType::InfiniteServer
87 */
88 helper infServerTransitionToolInfo() : PNML::ToolInfo {
89     return object PNML::ToolInfo {
90         tool := CONST::ToolInfoConstants::toolName.toString();
91         version := CONST::ToolInfoConstants::toolVersion.toString();
92         toolInfoGrammarURI := CONST::ServerType::InfiniteServer.toString().createURI();
93         formattedXMLBuffer := ("<value grammar=\"" +
94             CONST::ServerType::InfiniteServer.toString() + "\"/>").createLongString();
95     };
96 }
97
98 mapping OclAny::trace(to : OclAny, text : String) : TRACE::Trace {
99     init {
100         result := object TRACE::Trace {
101             fromDomainElement := self.eObject();
102             toAnalyzableElement := to.eObject();

```

```

100     rule := text;
101   }
102 }
103 }

```

### 3.1.5 Validation of the Transformation Patterns for the DPIM Level

The model transformations developed for the DPIM layer have been validated with the POSIDONIA Operations case study [14], within the approach in [18]. In particular, the approach – summarized by the Algorithm 1 – aims at deriving a GSPN model amenable to be used for performance predictions.

The input specification consists of: (i) a UML-based design that includes a (set of) Activity Diagram(s)  $\mathcal{AD}$ , which represent the execution process(es) of a data-intensive application – such as the parsing and the complex event processing (CEP) engine scenarios of POSIDONIA shown in Figures 4 and 5, respectively – and a Deployment Diagram<sup>1</sup>  $\mathcal{DD}$ , which specifies the software component allocation on computing nodes – such as the DD of Figure 6 – and (ii) the data log  $\mathcal{L}$  which includes a set of process execution traces.

The model transformations have been used in the first step (Step 1), where a Generalized Stochastic Petri Net (GSPN) model  $\mathcal{N}$  is automatically derived from each  $\mathcal{AD}$  and the  $\mathcal{DD}$  [1]. The Activity Diagrams (AD) and the Deployment Diagram (DD) in Figures 4, 5 and 6 are annotated with the DICE profile; in particular, input parameters are assigned to the mean durations of the action steps (i.e., *host-Demand* tagged-values) and to the data stream arrival rate (i.e., *arrivalRate* tagged-value). Figure 7 on page 30 shows the two GSPN subnets, in the dotted rectangles, that are derived via M2M transformation from the parsing scenario of Figure 4 and the CEP scenario of Figure 5, considering the logical resource restrictions specified in the DD of Figure 6 (*poolSize* tagged-values).

The next steps of the Algorithm 1 (Steps 2-7) consists in pre-processing the data logs  $\mathcal{L}$  and applying process mining techniques to: (i) check the conformance of the UML-based design with the data logs, and (ii) assign values to the rate parameters of the performance GSPN model. In particular, the data logs of POSIDONIA were collected in separate .csv files, 4 files related to the parsing process -one for each parser thread- with a mean number of 69 920 traces, and one single file related to the CEP process with a total of 56 698 traces. Each parsing trace represents the transformation of an NMEA message<sup>2</sup> – from the AIS (*Automatic Identification System*) receiver of the Balearic Islands – into an AIS sentence and includes 8 event occurrences, which correspond to the start and completion of each action modelled in the AD of Figure 4. Each trace of the CEP process represents instead the message handling by the CEP of the Palma port.

---

#### Algorithm 1 Approach

---

**Require:** UML design ( $\mathcal{AD}$ ,  $\mathcal{DD}$ ), data log ( $\mathcal{L}$ )

**Ensure:** Performance model ( $\mathcal{GSPN}$ ) & results ( $\mathcal{R}$ )

- 1: Get a normative model  $\mathcal{N}$  from  $\mathcal{AD}$
  - 2: Pre-process data log to get event log  $\mathcal{EL}$
  - 3: **repeat**
  - 4:   Filter  $\mathcal{EL}$
  - 5:   Check for conformance  $\mathcal{N}$  and  $\mathcal{EL}$
  - 6: **until** fitness  $\geq thres$
  - 7: Enhance  $\mathcal{N}$  with timing perspective:  $\mathcal{GSPN}$
  - 8: Performance analysis with  $\mathcal{GSPN}$ :  $\mathcal{R}$
- 

<sup>1</sup>As mentioned in Section A.1, *Deployment Diagrams* can be used to complement behavioral or structural models (among other uses). Since they are only used to complement behavioral diagrams from the transformations to analysis models point of view, we have not gone into detail for the sake of brevity.

<sup>2</sup>NMEA stands for *National Marine Electronics Association*, a US-based marine electronics trade organisation setting standards of communication between marine electronics.



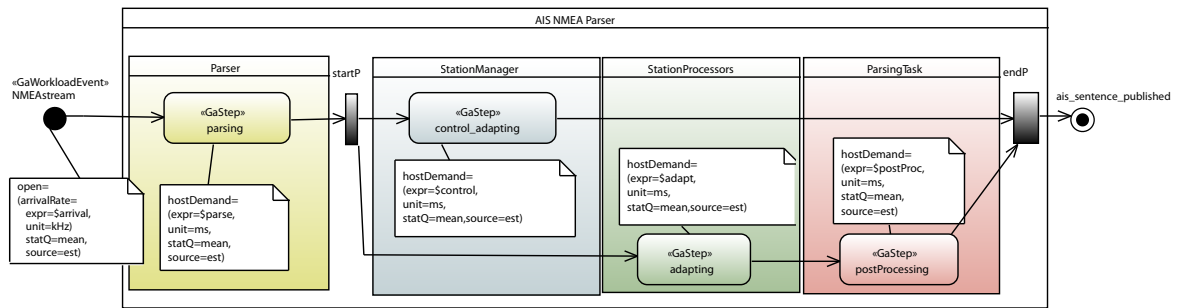


Figure 4: Activity Diagram for the Parsing scenario

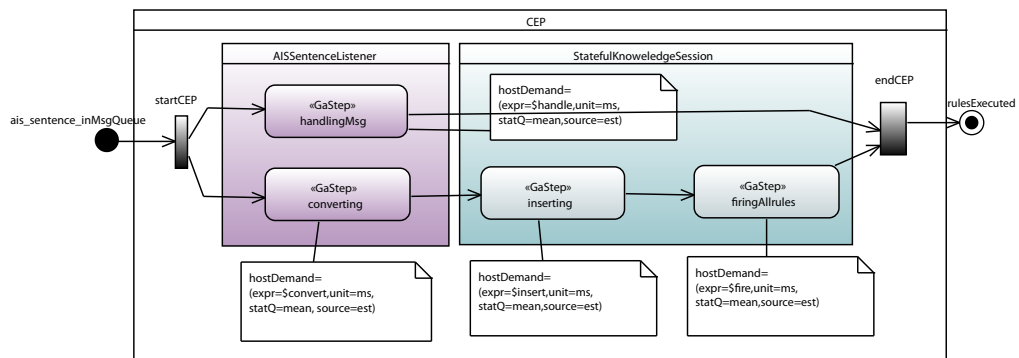


Figure 5: Activity Diagram for the Complex Event Processing engine scenario

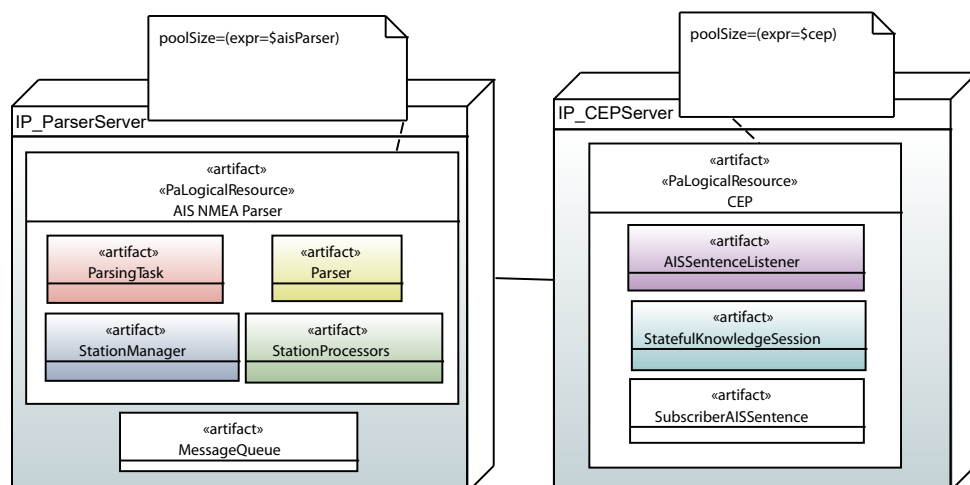


Figure 6: Deployment Diagram

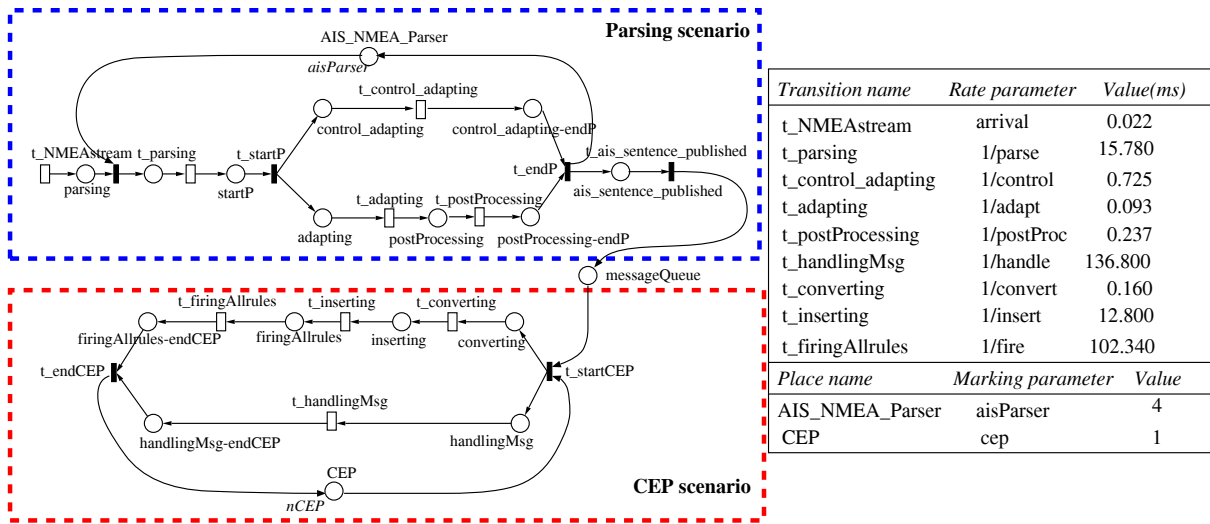


Figure 7: corresponding Generalized Stochastic Petri Net model

The values set to the rate parameters associated to the timed transition of the GSPN model were estimated by replaying the execution traces in the logs on the GSPN models derived from M2M transformation: the trace-driven simulator of the ProM tool [19] was used for this purpose. The table in Figure 7 (right side) shows the mean values obtained for the timed transitions of the two GSPN subnets of POSIDONIA (left side). All the transition firing times of the parser scenario subnet are characterized by the negative exponential distribution, since the standard deviations were similar to the mean values. Concerning the CEP scenario subnet, all the transition firing times are exponentially distributed but  $t_{firingAllrules}$  that is approximated by an Erlang distribution with  $k = 3$  steps.

The two GSPN subnets were validated separately by considering four parser threads and a single CEP (last two rows of the table in Figure 7), and the mean processing time as performance metric of reference. Both the analytical solver and the event driven simulator of GreatSPN [13] were used for this purpose. The relative error of the mean processing time of the parsing with respect to the one inferred by the logs was less than 1%. The relative error of the mean processing time of the CEP was around 10% (probably due to the abstraction level of the model, where the activation and firing of single CEP business rules are not explicitly represented).

The validated GSPN model (Figure 7) was used to evaluate the scalability of the POSIDONIA application considering different assumptions on the deployment environment. In particular, the deployment configurations included the number of parser threads (related to the number of AIS receptors), the number of CEPs for each geographical area and the arrival time of the data stream. Details on the performance analysis results can be found in [18].

## 3.2 Transformations to Quality Analysis Models for the DTSM Level

In this section, we focus on the transformations of UML models at DTSM level for two Big Data technologies: Apache Hadoop MapReduce and Storm. The transformation process of annotated UML diagrams to performance models requires an UML activity diagram for representing the logic and the temporal information of the application; and an UML deployment diagram for showing the number of available computational resources and their assignation to the different Big Data operations.

### 3.2.1 Hadoop MapReduce

This section is devoted to the Apache Hadoop MapReduce technology and it contains three subsections. The first one presents the UML models annotated with the DICE profile that capture the main concepts introduced in the Background section for an Apache Hadoop MapReduce framework. The second one details our approach of how a Hadoop UML model is transformed into a performance model. The third

one is devoted to the validation of our approach. To this end, we compare the results obtained by the execution of a Hadoop application in a controlled environment and the predictions of the performance model obtained by the transformation of the UML model that represents such application.

## UML Models for Hadoop MapReduce

In this section, we present our proposal for modeling Hadoop MapReduce applications with UML diagrams and the DICE profile. The focus of our proposal is on performance evaluation, which means that the DICE profile is used to introduce performance parameters (e.g., workload or host demands) in the UML models that represent the Hadoop MapReduce framework.

Figure 8 on the next page shows the UML activity diagram for an example of Hadoop application. The diagram starts with a mapping phase and finishes with a reducing phase. In this example, there are three classes of mappers and three classes of reducers, a capacity job scheduler and a specific workload. The number of mappers and reducers, the type of scheduler and the kind of workload are parameters that change from one configuration to another, but the UML activity diagrams will always have the same structure. These parameters of the Hadoop MapReduce cluster are captured by the DICE::DTSM::Hadoop profile annotations (i.e., stereotypes and its corresponding tags). In Figure 8 they appear as notes to ease its readability. Values using the symbol dollar (\$) represent variables. They are useful for parametrizing the UML model and consequently the resulting performance model.

Table 4 on page 33 links the Hadoop MapReduce concepts with the DICE::DTSM::Hadoop profile annotations. The DICE::DTSM::Hadoop profile includes five new and genuine stereotypes that are created for representing the schedulers, the workload, the mappers, the reducers and the cluster. The stereotypes and annotations for Hadoop are based on MARTE [15], DAM [20], the DICE::DPIM and Core profiles [3]. The DICE::DTSM::Hadoop stereotypes inherit or refine information from the mentioned profiles and also add new information. For instance, part of the annotations for the mapper and reducer stereotypes use MARTE-DAM for including temporal information (i.e., host demands) in the UML models. In the following we describe these five stereotypes.

The *workload* is described by the «*HadoopWorkloadEvent*» stereotype through two tags: *hadoopPopulation* and *hadoopExtDelay*. It defines the number of jobs for each class that are initially in the system; and the arrival rate of each class of job. The number of jobs is specified by the *hadoopPopulation* tag, i.e., an array of integers. The element \$nCi of the array represents the number of jobs of the class *i*. The arrival rate of jobs is specified by the *hadoopExtDelay* tag, i.e., an array of integers. The element \$thi of the array represents the time between the arrival of a new job of the class *i* to the cluster and the next one.

The stereotype «*HadoopScenario*» includes information of the complete cluster. For instance, the *scheduler algorithm* of the tasks and the *response time* of a job. The scheduler of the Hadoop MapReduce cluster is determined by the *jobSchedule* tag. It has an enumerable value that can be any of three common schedulers ({capacity, fifo, fair}). The response time (*respT*) measures the elapsed time since a user submits a job to the cluster until it returns the result. In other words, it includes the execution time of the map and reduce phases plus the time spent in the queues and communication delays. In fact, the response time is a metric defined within the «*HadoopScenario*» stereotype (see Figure 8). The mean (*statQ=mean*) response time is computed by the simulation of the performance model (*source=calc*) according to the information stored in the map (reduce) stereotypes and the rest of the diagram. The data type of the field *source* in the *hostDemand* tag is an enumerable {*est*, *calc*, *meas*, *req*}, where *est* means *estimated*; *calc* is *calculated*; *meas* is *measured* and *req* is *required*. The annotation *est* is equivalent to *meas*. They indicate that the execution time is estimated or measured by the user, and provided as input parameter to the DICE Simulation tool. The annotation *calc* is used for defining the metrics that will be computed in the performance model. The annotation *req* represents a temporal constraint that must be accomplished by the component (e.g., imposing a maximum delay for a response time). The DICE Simulation tool computes a metric, and then compares the result with the requirement for knowing if the requirement is satisfied or not.

Together with the «*HadoopScenario*» stereotype, we use the «*GaAnalysisContext*» stereotype from MARTE profile for summarizing all the parameters of the Hadoop MapReduce model. This stereotype

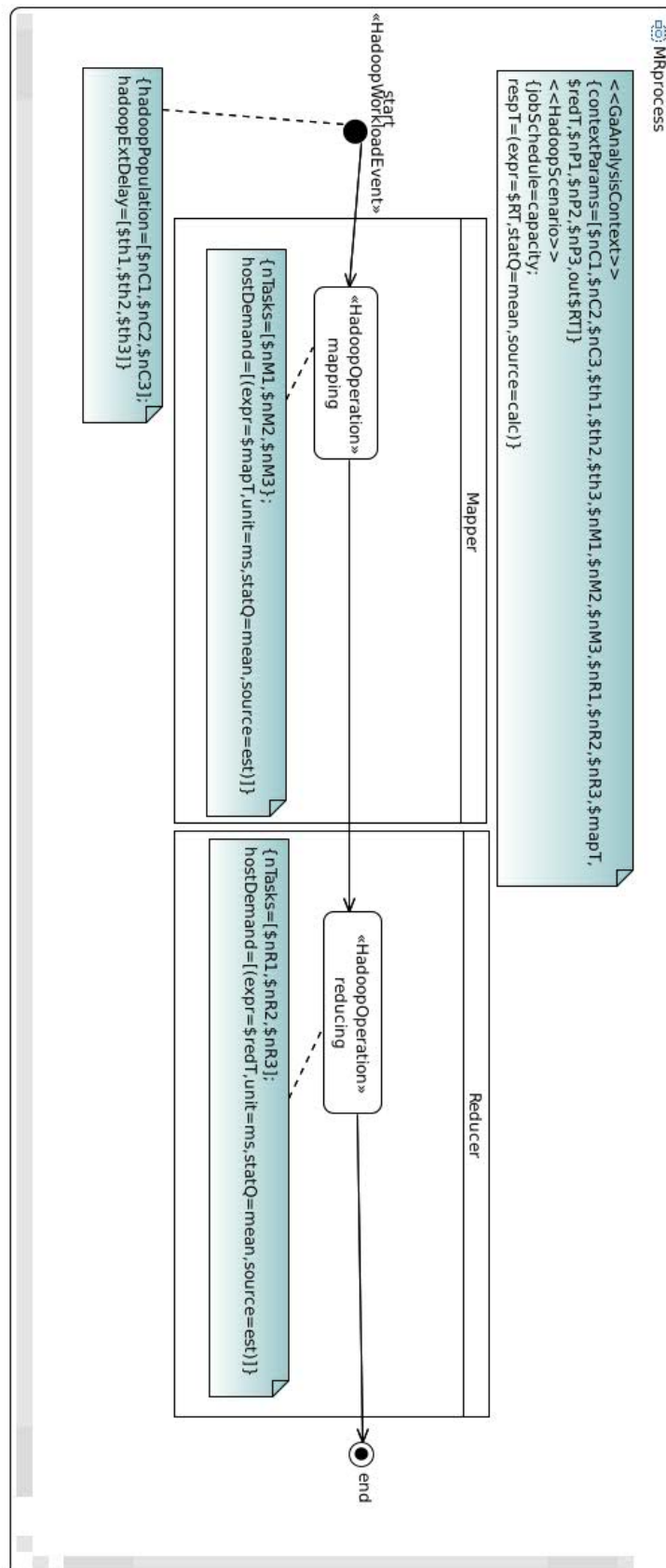


Figure 8: Example of an activity diagram for Hadoop MapReduce with DICE profile annotations

Table 4: DICE profile extensions for Hadoop MapReduce

HADOOP CONCEPTS	STEREOTYPE	MODEL ELEMENT (DIAGRAM)	TAG	DESCRIPTION	VALUE (EXAMPLES)
Job workload	« <i>HadoopWorkloadEvent</i> » inherits from « <i>GaWorkloadEvent</i> » (MARTE::GQAM)	initial node (AD)	hadoopPopulation (Array of NFP_Integer) hadoopExtDelay (Array of NFP_Duration)	Specifies the arrival pattern of each job class, in particular the population of Hadoop jobs and the think time (hadoopExtDelay).	hadoopPopulation = [ \$nC1, \$nC2, \$nC3 ] hadoopExtDelay = [ \$th1, \$th2, \$th3 ]
Scheduling policy and metric definition	« <i>HadoopScenario</i> » inherits from « <i>GaScenario</i> » (MARTE::GQAM)	activity diagram (AD)	jobSchedule (enumeration type) respT (NFP_Duration)	Specifies the job scheduling policy, and defines the response time metric that will be calculated by the simulation tool.	jobSchedule = capacity (value = \$mapT, statQ = mean, source = calc)
Map or Reduce tasks	« <i>HadoopOperation</i> » inherits from « <i>GaStep</i> » (MARTE::GQAM)	action node (AD)	nTasks (Array of NFP_Integer)	Specifies the number of map/reduce tasks for each job class.	nTasks = [ \$nM1, \$nM2, \$nM3 ]
			hostDemand (NFP_Duration) from « <i>GaStep</i> » (MARTE::GQAM)	Specifies the processing time of the map/reduce tasks. All the map/re- duce tasks represented by the stereotyped el- ement are assumed to have the same duration.	hostDemand = ( expr = \$mapT, statQ = mean, source = est)
Cluster partition	« <i>HadoopComputation- Node</i> » inherits from « <i>CoreComputationN- ode</i> » (DICE::DTSM::Core)	node (DD)	resMult (NFP_Integer) nCores (Array of NFP_Integer)	Specifies the number of cores assigned to each job class submitted to the map/reduce process- ing.	resMult=(expr=\$nP1) nCores = [ \$nP1, \$nP2, \$nP3 ]

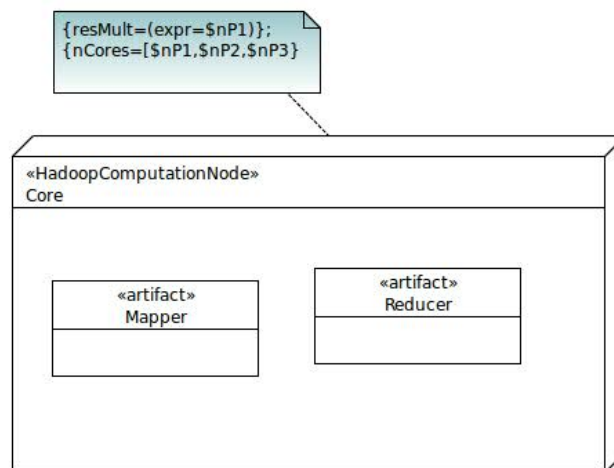


Figure 9: Example of a deployment diagram for Hadoop MapReduce with DICE profile annotations

is used for technical reasons. Mainly, it is used for saving the information of the Hadoop MapReduce configuration in order to simplify the transformation process from UML diagrams to performance models.

The *mappers* are functions that divide and preprocess the data during the mapping phase. The composition of the intermediate results is done by reduce functions (i.e., *reducers*) in the reducing phase. In an abstract sense, mappers and reducers are operations in the Hadoop cluster that execute a particular function during a certain amount of time. For that reason, they are both annotated with the same «*HadoopOperation*» stereotype. That is, the «*HadoopOperation*» stereotype models any operation in the Hadoop cluster. It includes a *hostDemand* tag for indicating the execution time of the function (field *value*). The execution time of the map (reduce) operations has been estimated before the simulation of the performance model (field *source=est*). This value represents the mean execution time (*statQ=mean*) of the map (reduce). The unit time is milliseconds (*unit=ms*). The stereotype also includes the tag *nTasks* (i.e., an array of integers) for specifying the number mappers (reducers) functions in which a class of job is divided. By default, we use \$nMi for naming the number of mappers and \$nRi for naming the number of reducers of the job class  *Ci*.

Finally, the «*HadoopComputationNode*» stereotype defines the physical assignation of map (reduce) functions to computational cores during the execution of the system (see the UML deployment diagram of Figure 9). The stereotype includes an array of integers *nCores* that associates \$nP<sub>i</sub> computational resources (cores) to each class of job ( *Ci*). The tag *resMult* defines the maximum number of resources that are available in the computational node, in this case, *resMult* = \$nP<sub>1</sub>.

## Hadoop MapReduce Transformations

A Hadoop MapReduce application, specified at DTSM level during the design phase using the UML activity and deployment diagrams with DICE::DTSM::Hadoop profile annotations, needs to be transformed into a formal model for assessing the performance and reliability of the designed system. The performance model that we use as target transformation of the UML diagrams is a Stochastic Well-formed colored Net (SWN) [21], i.e., a Petri net with a temporal interpretation and data types. A SWN is a useful formalism for the modeling and performance analysis of Hadoop MapReduce applications. Places represent the intermediate steps of the processing. Transitions represent the execution of map/reduce operations that are fired when certain conditions are met or a temporal delay is reached. Besides, tokens represent different type of elements depending on the color. Colors are used for distinguishing 1) the different users of the system for the scheduling policies, 2) the kind of processing according to the type of task (i.e., map or reduce), or 3) the computational resources (e.g., the cores assigned to a type of user). In summary, a SWN is an useful formalism for the modeling and performance analysis of Hadoop systems. Tools such as GreatSPN [13] allow the simulation and analysis of SWNs.



The transformation of the UML diagrams into a SWN must take into account all the information contained in the annotations (stereotypes and tags). The UML activity diagram is transformed into a single SWN. Later on, the annotations in the UML deployment diagram parametrize some parts of this SWN. We start with the transformation of the DICE::DTSM::Hadoop profile into an initial version of Petri net and show how to refine it according to the annotations. Figure 10 shows the transformation of the Hadoop phases (i.e., schedule, map and reduce) into a simple SWN that will be successively refined. A Hadoop MapReduce application is always transformed to the same Petri net prototype. The Petri net starts with the scheduling of the arriving tasks, continues with the mapping and reducing phases, and it finishes with the release of the resources. The Petri net is closed for emulating the batch processing mode of the Hadoop cluster. The SWN continues with the next batch once it finishes the current execution. The annotations of the stereotypes representing the workload, the scheduler, the mappers, the reducers and the cluster will only change the parameters of the net or a portion of it. That is, the information contained in the stereotypes mainly define the number of colors in the system (i.e., the number of jobs of each class in the net) or the performance information (i.e., temporal delays in the transitions).

The classes of jobs and the computational resources in the UML models are transformed to different colors in the SWN. In this example, there are three classes of colors for a *Job* data type. A *Job* must belong to any of the classes  $C_i$ . The notation  $u$  means that the *Job* type is the union of classes  $C_1$ ,  $C_2$  and  $C_3$ . There are a maximum of  $\$nCi$  tokens (jobs) of class  $C_i$  and a minimum of 1. The value  $\$nCi$  matches with the value of the element of the array *hadoopPopulation* in the «*HadoopWorkloadEvent*» stereotype.

The computational resources are partitioned according to the number of classes of jobs. There are three possible colors for the *Partition* data type. The type of job  $C_i$  has reserved a  $P_i$  number of cores. The value  $\$nP_i$  matches with the value of the corresponding element of the array *nCores* in the «*HadoopComputationNode*» stereotype. The place *Core* in the SWN indicates the total number of resources in the system. It is initialized with *numCores* tokens. In the SWN syntax,  $\langle S \rangle$  is the standard notation for the whole place color domain. That is,  $numCores = \$nP_1 + \$nP_2 + \$nP_3$ . The annotation *resMult* of the «*HadoopComputationNode*» is not explicitly used in the SWN. The SWN algorithm controls the assignation of resources of type  $P_i$  to jobs of class  $C_i$ .

In the SWN, the timed transitions fire following an exponential distribution probabilistic delay and in one case they are executed by infinite available servers. They determine the rate of messages per unit time. The arrival rate of new jobs is simulated by the think rate of the initial timed transition *extDelay*. The transition *extDelay* should be divided into three concurrent transitions for each type of job in the Figure 10 because the arrival rates were different, but we draw a single transition in the SWN for readability. The temporal information of the transition *extDelay* corresponds to *hadoopExtDelay* annotation of the stereotype «*HadoopWorkloadEvent*», and it has the value  $1/\$thi$ .

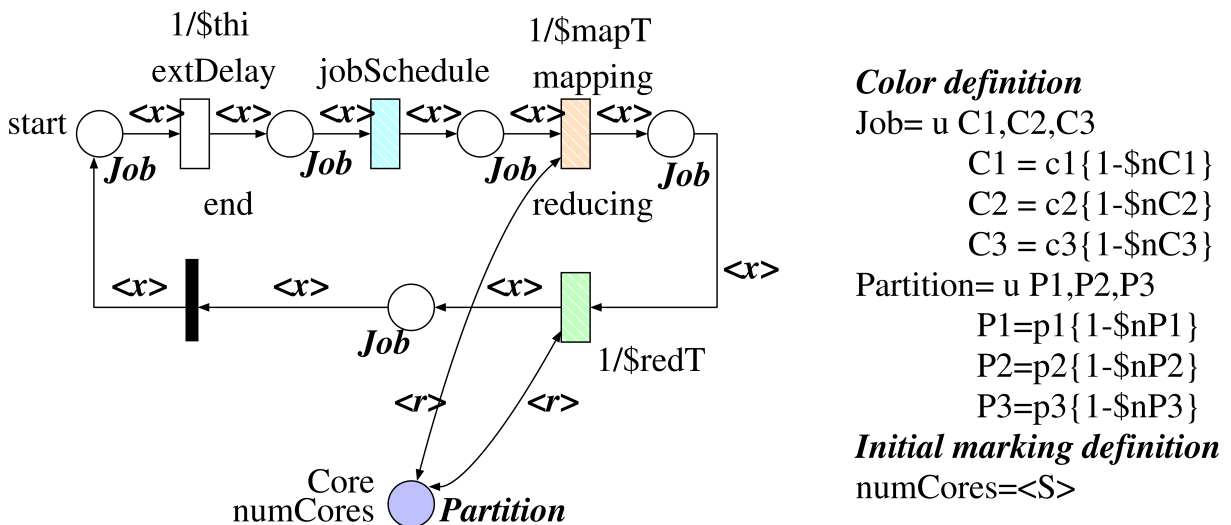


Figure 10: Petri net of Hadoop MapReduce

The timed transitions *jobSchedule*, *mapping* and *reducing* are abstract representations of the scheduling algorithm, the mapping and reducing phases. The *jobSchedule* transition will be analyzed and refined in the following paragraphs. The value  $\$mapT$  ( $\$redT$ ) assigned to the *mapping* (*reducing*) transition is the time required by the mapping (reducing) phase for creating an intermediate value and composing the final results. They correspond to the *hostDemand* tag annotation in the UML activity diagram. We use the inverse  $1/\$mapT$  ( $1/\$redT$ ) for expressing the throughput or the number of output tokens produced by unit time.

A more refined SWN is obtained by replacing the timed transitions (i.e., *jobSchedule*, *mapping* and *reducing*) by a concrete subnet capturing the specific characteristics of schedulers, mappers and reducers. For instance, Figure 11 shows a subnet that refines the map (reduce) transitions. This subnet divides a job into multiple tasks during the mapping (reducing) step. The fork (join) immediate transitions produce (consume) as many tokens (i.e., pairs of related  $\langle job, task \rangle$ ) as the cardinality of the static subclass  $T_i$ . The cardinality of the data type  $T_i$  is the number of subtasks (i.e.,  $\$nMi$  mappers or  $\$nRi$  reducers in the «*HadoopOperation*» stereotype) a job of type  $C_i$  is divided during the map (reduce) phase. The cardinality of  $T_i$  is denoted by  $\langle S T_i \rangle$ .

The arcs of the SWN are annotated with the color of the tokens that can traverse them. For example, the label  $\langle x \rangle$  is used for expressing the type of job,  $\langle t \rangle$  is the type of task and  $\langle r \rangle$  is the type of resource. The colors in the arcs also impose conditions for firing a transition of the SWN. Besides, they show the assignation and release of resources. For instance, the *taskScheduling* transition in Figure 11 has two input arcs ( $\langle x, t \rangle$  and  $\langle r \rangle$ ), and an output arc ( $\langle x, t, r \rangle$ ). It means that a resource  $\langle r \rangle$  is taken from the place *Core* and assigned to the task  $\langle t \rangle$  of the job  $\langle x \rangle$  producing a tuple  $\langle x, t, r \rangle$ . Internally, the *taskScheduling* transition checks the values of the data types of the input arcs and guarantees that only an output token is produced when certain color conditions are met. For example, a firing condition is that the color of the resource  $\langle r \rangle$  and the job  $\langle x \rangle$  must be compatible. That is,  $\langle r \rangle$  belongs to the partition  $P1$  and the job  $\langle x \rangle$  belongs to the job class  $C1$  (i.e.,  $P1$  represent the set of cores reserved for the jobs of class  $C1$ ).

The scheduling policy generates different configurations during the transformation phase to the performance and validation model. The subnet that will replace the *jobSchedule* transition in the Figure 10 will change depending on the scheduling policy adopted. Figure 12 on page 38 is an adaptation of the initial SWN that is able to capture completely the behavior of the *capacity* scheduler policy. For simplicity, there is only one type of jobs (users) in the cluster in this example (i.e., class  $C1$ ). The *capacity* scheduler will generate an independent FIFO queue for each job class.

To enforce the FIFO scheduling, each job is assigned an identifier  $ID_i$ . The initial marking  $M2$  of the place  $IDs1$  is set to the first index (1) of the color class  $ID$ . Once transition *think* sends a job to the ready state, it increases this index by one ( $i$  is equivalent to  $i := (i+1) \bmod C1$ ). The transition *generateMaps* will start the job  $\langle x, i \rangle$  that has the index equal to the one it is getting from place  $IDs2$  ( $\langle i \rangle$ ). In other words, the job that has its turn will start the Map phase. Whenever a job gets resources for all of its reduce tasks (place *wait4ResRed* drains), the job with the next index will be started thanks to the transition *startNext*, which updates the  $IDs2$  place with the next index. This condition is controlled by an inhibitor arch that activates *startNext* when *wait4ResRed* is empty. Observe that the map tasks  $\langle y, t \rangle$ , associated to a job  $\langle y \rangle$ , are generated when all the reduce tasks  $\langle x, t \rangle$ , associated to the previous job  $\langle x \rangle$ , are not waiting for resource availability.

When a job  $\langle x \rangle$  is ready to be processed and it has its turn—i.e., the place *jobReady* is marked with a token  $\langle x, i \rangle$  and the  $IDs2$  place is marked with the same index  $\langle i \rangle$ —a group of  $\$nM1$  map tasks are generated (firing of *generateMaps* transition). Such tasks, associated to job  $\langle x \rangle$  are represented by  $\$nM1$  pairs  $\langle x, t \rangle$ , where the color  $\langle t \rangle$  belongs to the subclass *Map* of the basic color class *Task*. Each task  $\langle x, t \rangle$  needs to acquire a resource  $\langle r \rangle$  to be executed (firing of *getResMap* transition). The map tasks are concurrently executed according to resource availability. The set of resources is again defined by the basic color class *Partition*, which in this case consists of a unique partition  $P1$  including  $\$nP1$  resources.

The timed transition *map* models the duration of the map task execution ( $1/\$mapT$ ) and in this case, its firing time is an Erlang-distributed random variable. The map stage is finished when all the map tasks  $\langle x, t \rangle$  associated to job  $\langle x \rangle$  have been executed. The firing of the *joinMaps* transition models the beginning of the next processing step, where  $\$nR1$  reduce tasks are generated. The reducing step is



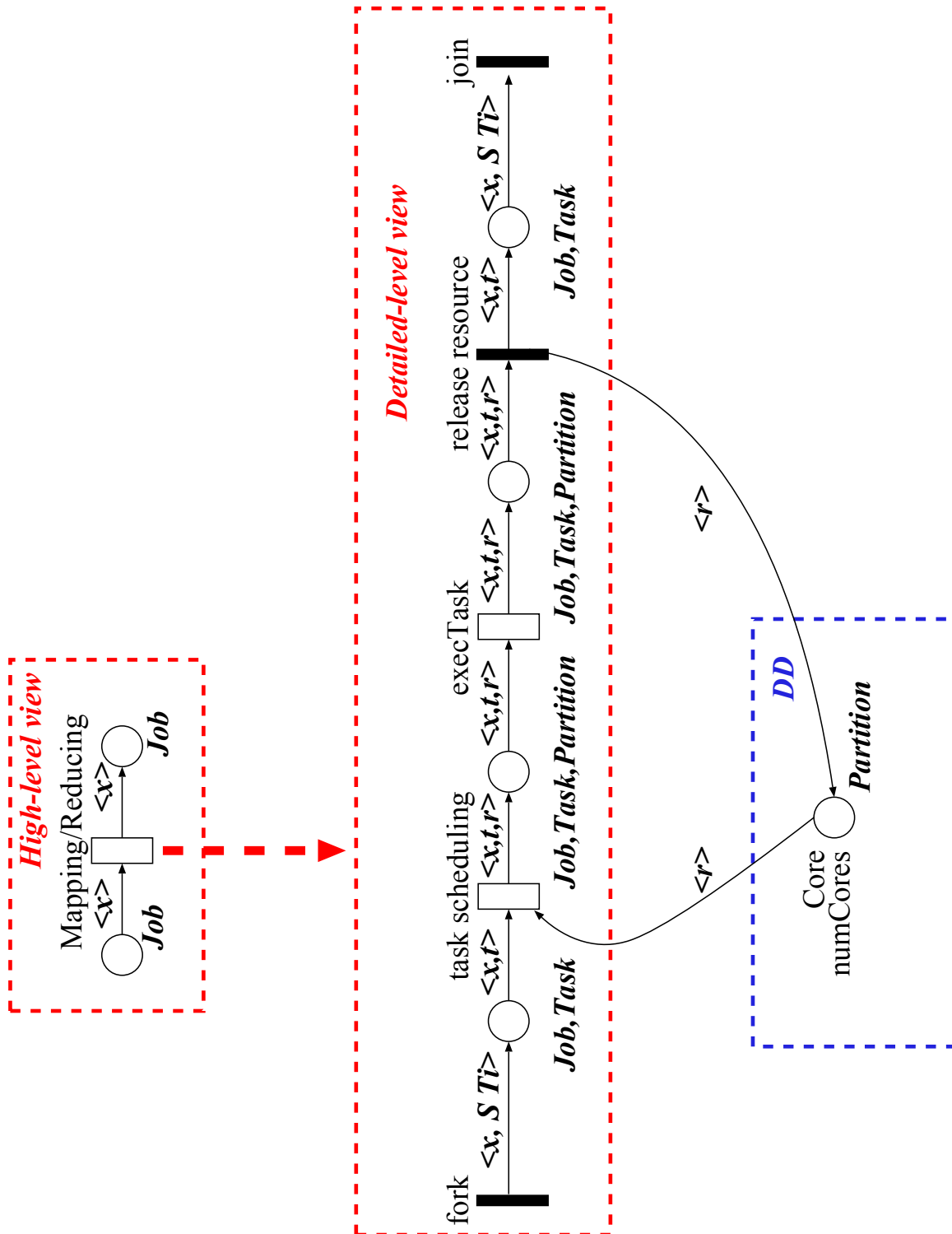


Figure 11: Mapping/reducing subnets of Hadoop MapReduce

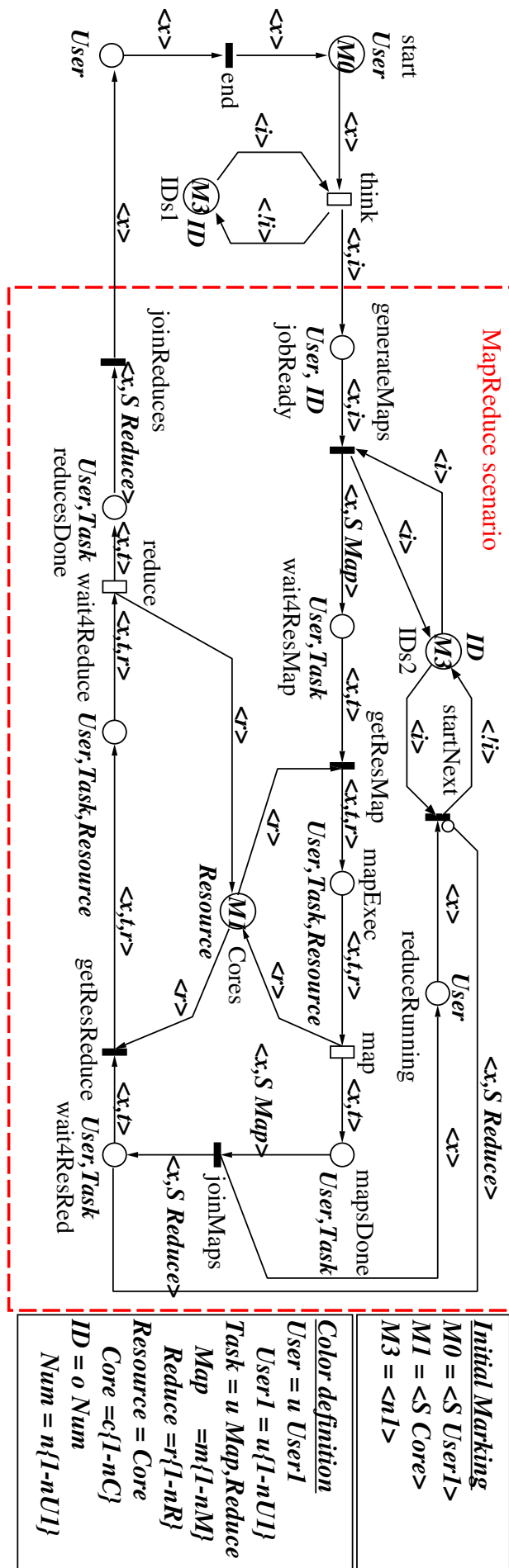


Figure 12: SWN model of Hadoop MapReduce with Capacity scheduler

similar to the mapping step (the reduce tasks are executed at a rate of  $1/\$redT$ ). The only difference is that the reduce tasks, associated to job  $\langle x \rangle$ , are represented by  $\$nR1$  pairs  $\langle x, t \rangle$  where the color  $\langle t \rangle$  belongs to the subclass *Reduce* of the basic color class *Task*.

## Hadoop MapReduce Validation

The objective of this section is to validate the results obtained by our performance models against real results. Consequently, we will validate our proposal for transforming UML models annotated with the DICE profile into a Petri net prepared for the performance evaluation of the modeled system.

The validation results for Hadoop MapReduce has been already presented in Section 6 of the DICE Deliverable 3.8 [22], but we also include a subset of them here in our deliverable in order to be self-contained. The results of the DICE Deliverable 3.8 has been taken from [23]. These results consider a refined SWN model with fault-tolerance capabilities.

The experiments are executed on Amazon EC2 and CINECA, the Italian supercomputing center. The target version was Hadoop 2.6.0. We have studied the performance for various configurations of the Hadoop MapReduce framework. That is, we have analyzed the simulation errors for different number of mappers (nMaps), reducers (nRed), cores and users. We have chosen a set of SQL queries that are translated into MapReduce jobs using Apache Hive [24]. They are executed over a dataset of several files ranging from 250 GBytes to 1 TByte that are used as external tables. The metric that we measured was the system response time. The GreatSPN tool [13] has been used for the simulation of the SWN model.

The validation results are summarized in Table 5. The SQL queries are identified by the name R1-R5 in the first column of the table. A description of the SQL sentences can be consulted in the DICE Deliverable 3.8. The last column of the table represents the percentage of relative error between the estimated and real response times. The response times obtained by the simulation of the SWN (column  $T_{swn} [ms]$ ) are close to the response times obtained by real executions (column  $T [ms]$ ) in the Hadoop MapReduce cluster for most situations. The simulation tool offers good estimations of the real execution

Table 5: Validation of the SWN for Hadoop MapReduce

QUERY	USERS	CORES	SCALE [GB]	NMAPS	NRED	T [ms]	T SWN [ms]	% ERROR
R1	1	240	250	500	1	55 410	50 629.58	-8.63
R2	1	240	250	65	5	36 881	37 976.82	2.97
R3	1	240	250	750	1	76 806	83 317.27	8.48
R4	1	240	250	524	384	92 197	89 426.51	-3.01
R1	1	60	500	287	300	378 127	330 149.74	-12.69
R3	1	120	750	1 148	1 009	661 214	698 276.75	5.61
R4	1	60	750	868	910	808 490	806 366.51	-0.26
R3	1	80	1000	1 560	1 009	1 019 973	1 020 294.84	0.03
R5	1	80	1000	64	68	39 206	38 796.47	-1.04
R1	3	20	250	144	151	1 002 160	909 217.89	-9.27
R1	5	20	250	144	151	1 736 949	1 428 894.40	-17.74
R2	3	20	250	4	4	95 403	99 219.94	4.00
R2	5	20	250	4	4	145 646	88 683.10	3.09
R1	5	40	250	144	151	636 694	613 577.53	-3.63
R2	3	40	250	4	4	86 023	119 712.30	-17.81

of Hadoop MapReduce systems.

The work described in this section of the document is the result of the analysis of Apache Hadoop MapReduce, which is considered as a reference technology. Most of the concepts can be adapted to other frameworks, since the model is very general. We have shown the transformation of UML activity and deployment diagrams to performance models that capture the essentials of the technology. The experimental results have confirmed the feasibility of our approach for common Apache Hadoop MapReduce configurations.

### 3.2.2 Storm

This section is devoted to the Apache Storm technology and it contains three subsections. The first one presents the UML models annotated with the DICE profile that capture the main concepts introduced in the Background section for an Apache Storm topology. The second one details our approach of how a Storm UML model is transformed into a performance model. The third one is devoted to the validation of our approach. To this end, we compare the results obtained by the execution of a Storm application in a controlled environment and the predictions of the performance model obtained by the transformation of the UML model that represents such application.

#### UML Models for Storm

In this section, we present our proposal to represent Storm topologies using UML diagrams and the DICE profile. The focus of our proposal is on performance evaluation, which means that the DICE profile is used to introduce performance parameters (e.g., host demands) in the UML models that represent the Storm topology.

Figure 13 shows the UML activity diagram for an example of Storm topology with two spouts and three bolts in a pipeline layout. Of course, more complex Storm topologies such diamonds or stars can also be expressed using UML activity diagrams. The UML activity diagram for Storm will always start with a set of initial nodes corresponding to spout elements because they are the sources responsible of inserting tuples in the topology at a certain speed. The parameters of the Storm application are captured by the DICE::DTSM::Storm profile annotations (i.e., stereotypes and its corresponding tags). In Figure 13 they appear as notes to ease its readability. Values using the symbol dollar (\$) represent variables. They are useful for parametrizing the UML model and consequently the resulting performance model.

It must be noticed that the usual interpretation of the UML activity diagram cannot be applied here because the rounded rectangles representing actions will never finish according to the characteristics of Storm as a stream processing technology. Besides, the arcs connecting actions do not represent a logical succession of actions but a communication channel between two processes (i.e., spouts and/or bolts). Hence, in our approach a UML activity diagram is interpreted as a DAG for a particular Storm topology.

Tables 6–7 link the Storm concepts with the stereotypes of the DICE::DTSM::Storm profile and the annotations. The DICE::DTSM::Storm profile includes three new and genuine stereotypes for representing the spouts, bolts, and the arcs between nodes. The stereotypes and annotations for Storm at DTSM level are based on MARTE [15], DAM [20], the DICE::DPIM and Core profiles [3]. The DICE::DTSM::Storm stereotypes inherit or refine information from the mentioned profiles and also add new information. For instance, part of the annotations for the spout and bolt stereotypes use MARTE-DAM for including temporal information (i.e., host demands) in the UML models. In addition, MARTE allows the annotation of the number of resources (Cluster field of Table 7) during the deployment.

*Spouts* and *bolts* are important concepts that have been modeled with stereotypes in the DICE profile for Storm. The components *spout\_1* and *spout\_2* of the UML diagram use the stereotype «*StormSpout*», and the components *bolt1\_1*, *bolt1\_2* and *bolt2\_3* use the stereotype «*StormBolt*». They have independent DICE stereotypes because they are conceptually different but spouts and bolts share the main core attributes inherited from «*CoreDAGNode*»/«*CoreDAGSourceNode*» from the Core profile of DICE. That is, the spout and bolt stereotypes include the definition of the level of parallelism and the execution time. On one hand, the parallelism of a component is specified by the value of the *parallelism* tag. It determines the number of threads executing the same component. On the other hand, each thread of a

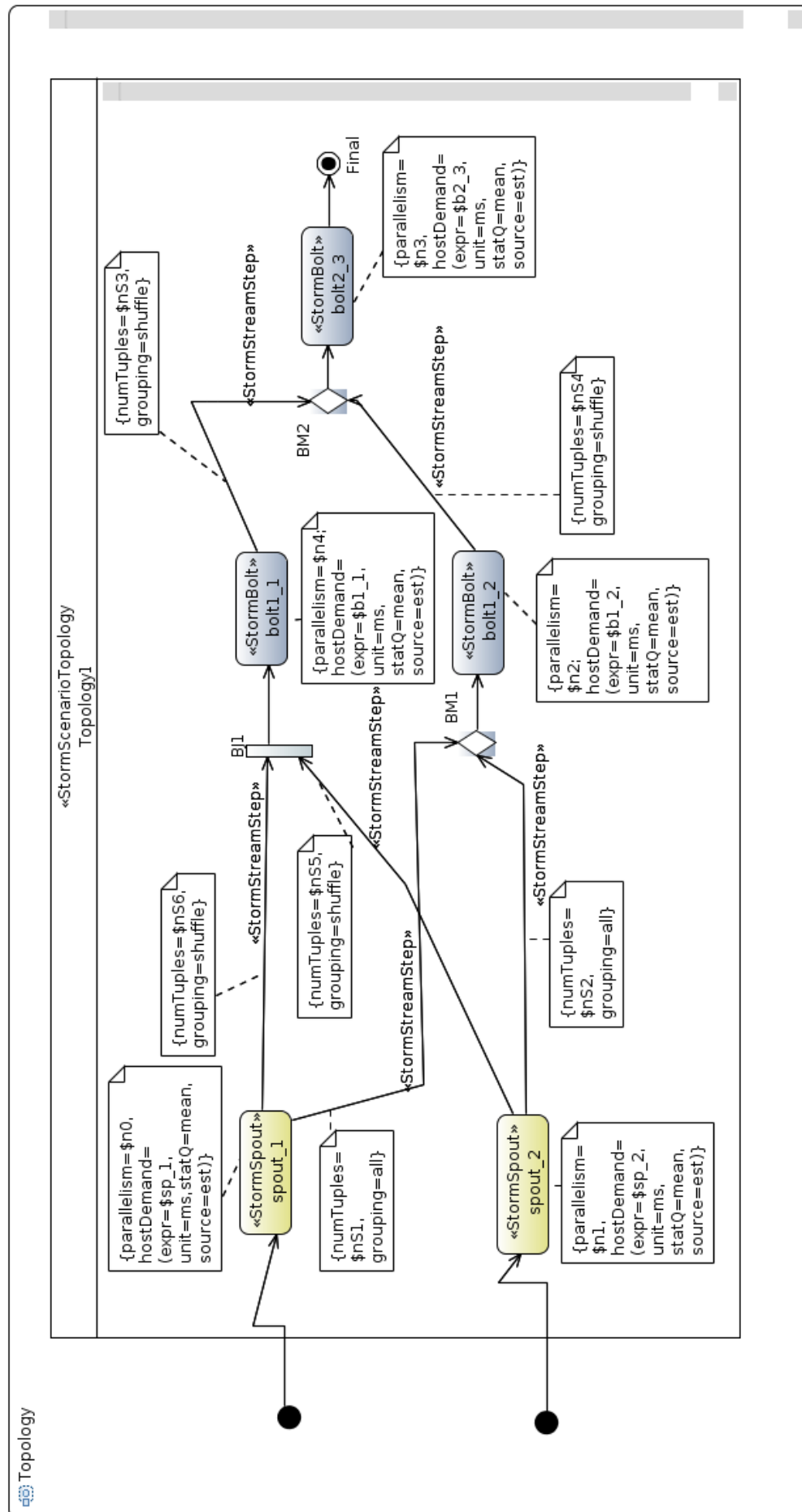


Figure 13: Example of an activity diagram for Storm with DICE profile annotations

Table 6: DICE profile extensions for Storm (1)

HADOOP CONCEPTS	STEREOTYPE	MODEL ELEMENT (DIAGRAM)	TAG	DESCRIPTION	VALUE (EXAMPLES)
Storm Spout	« <i>StormSpout</i> » inherits from « <i>CoreDAG-SourceNode</i> » (DICE::DTSM::Core)	action node (AD)	parallelism (NFP_Integer) from « <i>CoreDAGNode</i> » (DICE::DTSM::Core)	Specifies the number of threads associated to the spout/bolt.	parallelism=\$n0
Storm Bolt	« <i>StormBolt</i> » inherits from « <i>CoreDAGNode</i> » (DICE::DTSM::Core)		hostDemand (NFP_Duration) from « <i>GasStep</i> » (MARTE::GQAM)	Specifies the processing time of the Storm tasks. All the spout/bolt tasks represented by the stereotyped element are assumed to have the same duration.	hostDemand= (expr=\$sp_1, statQ=mean, source=est)

Table 7: DICE profile extensions for Storm (2)

HADOOP CONCEPTS	STEREOTYPE	MODEL ELEMENT (DIAGRAM)	TAG	DESCRIPTION	VALUE (EXAMPLES)
Stream	« <i>StormStreamStep</i> » <i>inherits from «GaStep»</i> <i>(MARTE::GQAM)</i>	edge between components (AD)	numTuples (NFP_Integer)	Specifies the number of tuples from the source that are necessary to produce an output tuple.	numTuples=\$nS1
			grouping (enumeration type)	Specifies the destination of the input tuple. It can be sent to all the threads of a component ( <i>all</i> ) or randomly to any of them ( <i>shuffle</i> ). More group- ing options are available in Storm, but only these two ones are considered for the beginning.	grouping=shuffle
Cluster partition	« <i>GaExecHost</i> » (MARTE::MARTE- _Analysis- Model::GQAM)	node (DD)	resMult (NFP_Integer)	Specifies the number of cores assigned to the Storm operations.	resMult=(expr=\$c1)

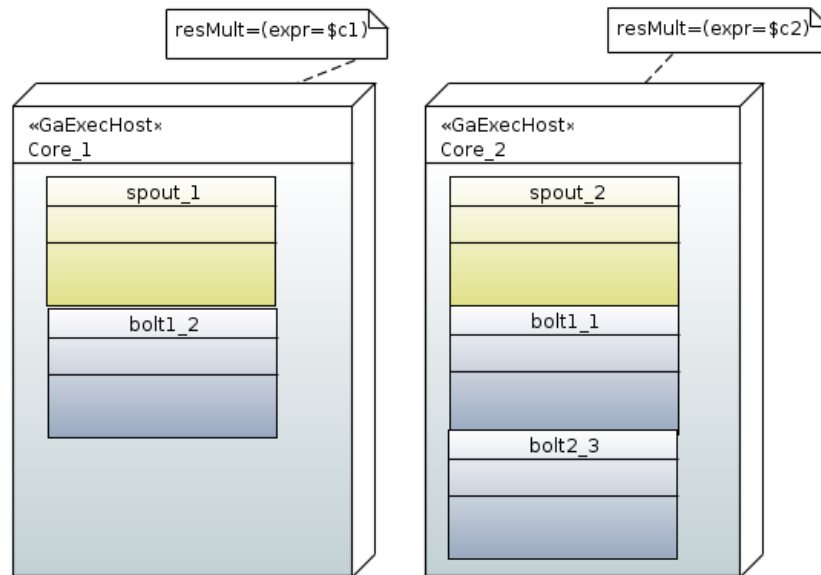


Figure 14: Example of deployment diagram for Storm with DICE profile annotations

component has an estimated execution time whose value is contained in the field *value* of the *hostDemand* tag. The field *source=est* indicates that the execution time is estimated in the real system before the simulation of the performance model. The explanation of the *source* field is located in the equivalent section of the Hadoop MapReduce transformations. The execution time represents the mean execution time (*statQ=mean*) for all the threads. The unit time is milliseconds (*unit=ms*).

The Storm concept of *stream* is captured by the stereotype «*StormStreamStep*», which is attached to arcs between actions in the UML activity diagram. This stereotype has two tags: *grouping* and *num-Tuples*. The former for specifying the policy {all,shuffle} and the latter for specifying the number of tuples that a node must consume for producing an output. The synchronization policies for composing the messages (AND/OR configurations in Figure 31) are represented graphically by diamonds (OR) and bars (AND).

Finally, Figure 14 shows the annotated UML deployment diagram. It maps the elements of the Storm topology into the hardware resources. The UML deployment diagram complements the UML activity diagram for the definition of the system parallelism. More in detail, the annotations of the UML deployment diagram collect the information of the number of computational resources available in the artifacts of the system; and represents the association of each thread of a component (spout or bolt) to a core or virtual machine. Similarly to Hadoop MapReduce, the *resMult* tag, from the «*GaExecHost*» stereotype, indicates the number of available cores or instances using the expression \$c1 (\$c2).

## Storm Transformations

The topology of a Storm application, specified at DTSM level during the design phase using the UML activity and deployment diagrams with DICE::DTSM::Storm profile annotations, needs to be transformed into a formal model for assessing the performance and reliability of the designed system. The performance model that we use as target transformation of the UML diagrams is a Generalized Stochastic Petri Net (GSPN) [25], i.e., a Petri net with a temporal interpretation. A GSPN is a useful formalism for the modeling and performance analysis of Storm systems. Places represent the intermediate steps of the processing. Transitions represent the execution of Storm operations and are fired when certain conditions are met or a temporal delay is reached. Besides, tokens represent the messages (tuples) sent between components; and arc weights determine the number of tokens (messages) required for firing the transition. No colors (i.e., *data types* in the Petri net) are considered for this moment, but they will serve in the future for defining different pipelines distinguishing 1) the kind of processing according to the type of data, or 2) the characteristics of the computational resources (e.g., location of cores, computational



speed, etc.).

The transformation of the UML diagrams into a GSPN must take into account all the information contained in the annotations (stereotypes and tags). The UML activity diagram is transformed into a single GSPN. Later on, the annotations in the deployment diagram parametrize some parts of the GSPN. We start with the transformation of the basic stereotypes of the DICE::DTSM::Storm profile into Petri nets and show how to compose them incrementally. Figure 15 shows the transformation of the core Storm components (i.e., spouts and bolts) into a GSPN. The example presents a single spout that creates tuples, sends them to the bolt, and the bolt processes them. Spouts and bolts stereotypes are always transformed to the same subnet prototype; the stereotype «*StormStreamStep*» will change the weights and configurations of some particular arcs and transitions. The Petri net starts with a set of spouts and ends with a set of bolts. The workload in the Petri net is open for emulating the streaming characteristic of Storm.

The timed transitions *execTask* are abstract representations of the spout and bolt functions. The tuples are generated and inserted into the pipeline at a certain rate according to the rate of the initial timed transition representing the spout. The value  $\$sp1$  ( $\$b1$ ) is the time required by the spout (bolt) for creating a new tuple. They correspond to the value of the *hostDemand* tag annotation in the UML activity diagram. In the GSPN, the timed transitions are configured as infinite available servers and constrained by the internal parallelism of the spouts (bolts). The specification of the timed transitions is provided by the values  $1/\$sp1$  for the spouts ( $1/\$b1$  for the bolts). They determine the rate of messages per unit time.

The number of threads assigned to a spout (bolt) is controlled by an explicit place named *spoutParallelism* (*boltParallelism*). It is initialized with the value indicating the degree of *parallelism* in the annotations (e.g.,  $\$n0$ ). A spout (bolt) starts running when the corresponding thread gains access to a computational resource (core).

The place *Core* in the GSPN defines the multiplicity of hardware resources (i.e., number of cores) according to the value of *resMult* tag in the UML deployment diagram. This value then represents the total number of available cores in the system ( $\$c1+\$c2$  in the case of the Figure 14). Places *spoutOutput* (*boltOutput*) represent just intermediate buffers for the communication between components. The output messages generated by a spout (bolt) are put in those places. For each stream connection, an independent place *spoutOutput* (*boltOutput*) is created.

In Figure 15, spouts and bolts are abstracted by timed transitions. Nevertheless, spouts and bolts are generic programs that execute code for creating or manipulating tuples. In some cases, it is interesting to increase the granularity of the GSPN for capturing the details of the execution flow of a spout (or bolt) program. Some of the advantages of Petri nets are the modularity and composition of the models in order to handle the level of granularity. Timed transitions can be replaced by new subnets representing

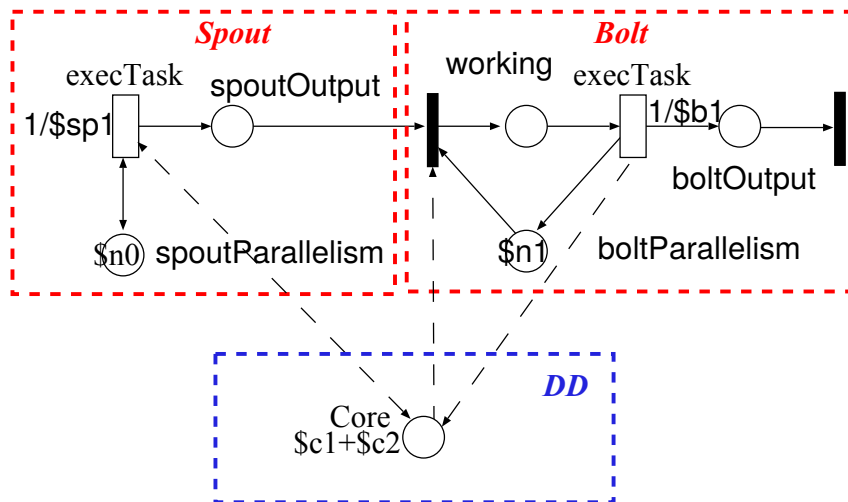


Figure 15: Petri net for spout and bolt components in Storm

the internal workflow of spouts and bolts functions with minor impact to the overall Petri net model.

Figure 16a–16b, detail a Petri net modeling the annotations of the Storm communication channel (i.e., the «*StormStreamStep*» stereotype). They detail the transformations for the attributes of *grouping*, *numTuples* and the type of *synchronization*. The temporal information of the timed transitions is omitted in the images for readability. The *grouping* and *numTuples* annotations change the arc weights in the GSPN; while the type of synchronization, represented by bars (*and* policy) and diamonds (*or* policy), varies the number of immediate transitions between the end of a component and the beginning of the next one.

The *grouping* tag indicates the number of threads required by the next component for processing the message. It defines the number of tokens (messages) that the timed transition *execTask* will put in the place *spoutOutput* (*boltOutput*) of the GSPN. If the value of *grouping* is *shuffle*, only a thread will receive the tuple: the arc in the GSPN has unitary weight. If the value of *grouping* is *all*, every thread will receive the tuple: the arc in the GSPN has the same weight as the value of *parallelism* in the next component. For instance, \$n2 in Figure 16a–16b represents the parallelism of the bolt.

The *numTuples* attribute indicates the number of input messages (tokens) that a bolt requires for creating a new message. It defines the number of tokens that the bolt will read from the place *spoutOutput* (*boltOutput*) of the previous component in the GSPN. For instance, the bolt requires \$nS1 tokens from the first spout and \$nS2 tokens from the second spout in Figure 16a–16b.

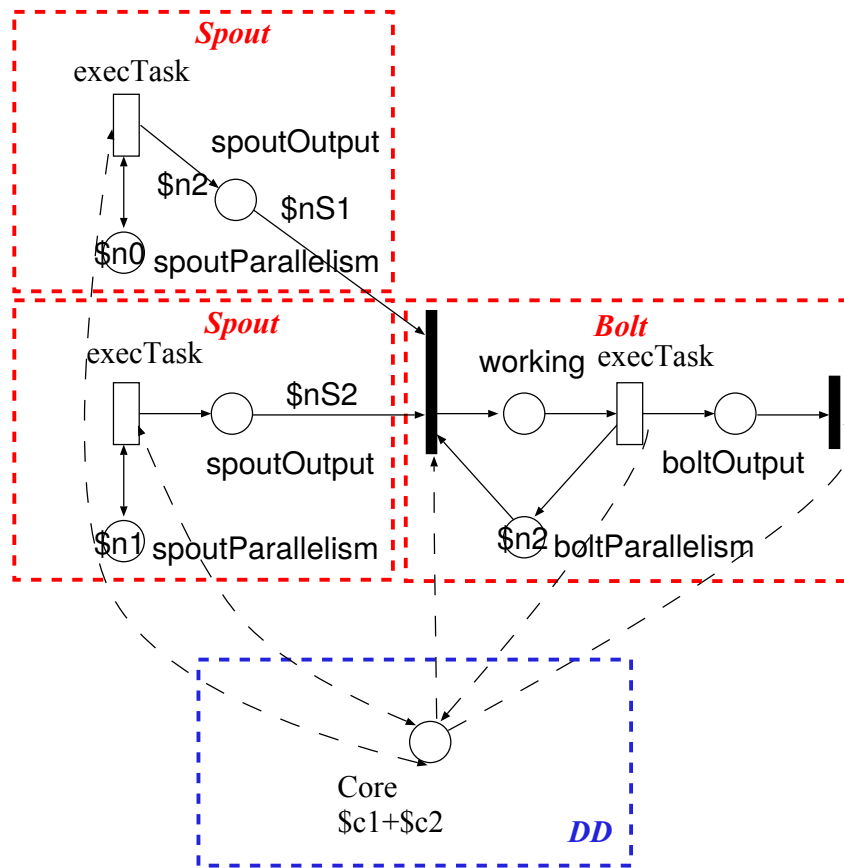
Joins (bars) in the UML activity diagram representing an *and* synchronization are transformed into a single immediate transition in the GSPN that receives the arcs from all the precedent components. Merges (diamonds) in the UML activity diagram representing an *or* synchronization are transformed into several immediate transition in the GSPN, one per input arc.

Figure 16a shows two spouts sending tuples to a bolt. The bolt requires \$nS1 tokens from the first spout, *and* \$nS2 tokens from the second spout in order to proceed. The connection between the first spout and the bolt follows an *all* policy, and the connection between the second spout and the bolt follows a *shuffle* policy. The parallelism is initialized with \$n0 and \$n1 threads for spouts and \$n2 threads for the bolt. Figure 16b shows the same GSPN model. However, in this case the bolt requires \$nS1 tokens from the first spout, *or* \$nS2 tokens from the second spout instead of receiving both \$nS1 *and* \$nS2 messages. Arcs between the core resources and the transitions are dotted for readability.

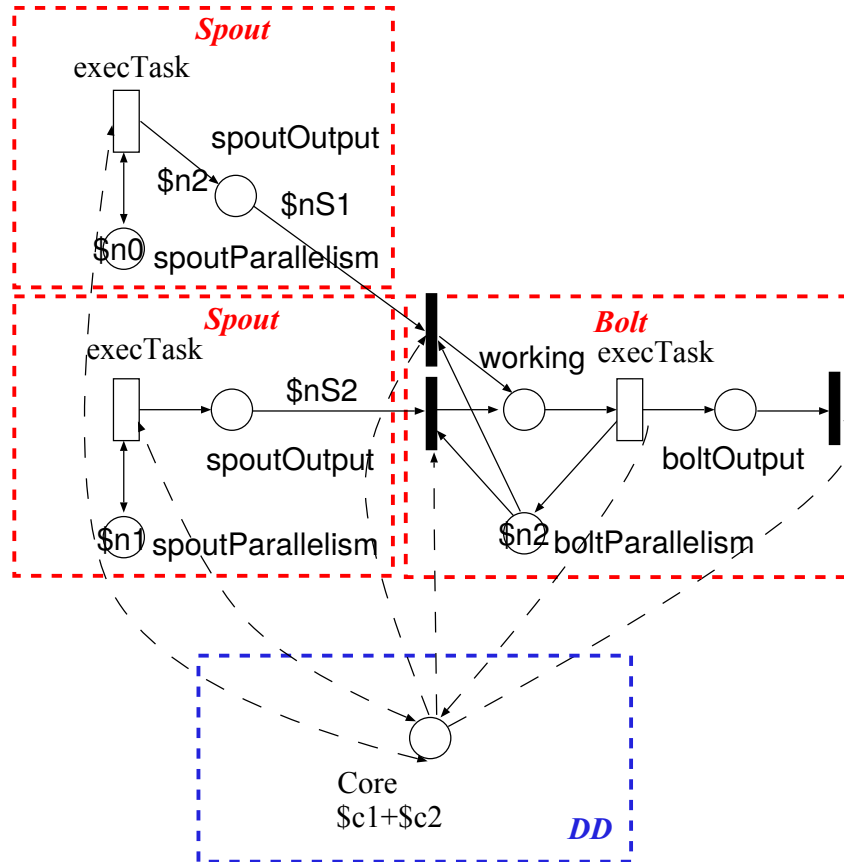
Finally, Figure 17 represents the complete GSPN modeling the main concepts of Figure 13–14. The GSPN model has been created according to the values of the annotations in the UML diagrams. The GSPN model includes the arc weights (*numTuples*, *grouping*), the level of parallelism (*parallelism*), the number of available cores (*resMult*), and the execution times (*hostDemand*). The names of the places and transitions are omitted for readability, but they can be clearly identified. The spout\_1 has parallelism \$n0 and the spout\_2 has parallelism \$n1. The bolt1\_1 requires \$nS6 messages from spout\_1 and \$nS5 messages from spout\_2 with *shuffle* policies. The bolt1\_2 requires \$nS1 messages from spout\_1 or \$nS2 messages from spout\_2 with *all* policies. The bolt2\_3 requires \$nS3 messages from bolt1\_1 or \$nS4 messages from bolt1\_2 with *shuffle* policies.

## Storm Validation

The objective of this subsection is to validate the results obtained by our performance models against the results obtained by deploying the real system in a real cluster of computers. Consequently, we will validate our proposal, presented in the previous sections, for transforming UML models annotated with the DICE profile into a Petri net prepared for the performance evaluation of the modeled system. To this end, we use the GSPN of Figure 17 on page 48 that we have obtained by the transformation of the annotated UML activity diagram of Figure 13 according to the transformations rules proposed. For getting consistent results, our validation applies different configurations depending on the number of available cores and computers in the cluster, the arrival rate of messages and the processing time of the intermediate components (bolts). We simulate the GSPN with the GreatSPN tool [13] and we compare the results with those obtained for the system deployed in a real cluster. We measure the relative error between real simulations and predictions ( $\text{abs}(\text{real-prediction})/\text{prediction}$ ).



(a) Petri net modeling the annotation parameters of the Storm communication channel (1)



(b) Petri net modeling the annotation parameters of the Storm communication channel (2)

Figure 16: Petri net modeling the annotation parameters of the Storm communication channel

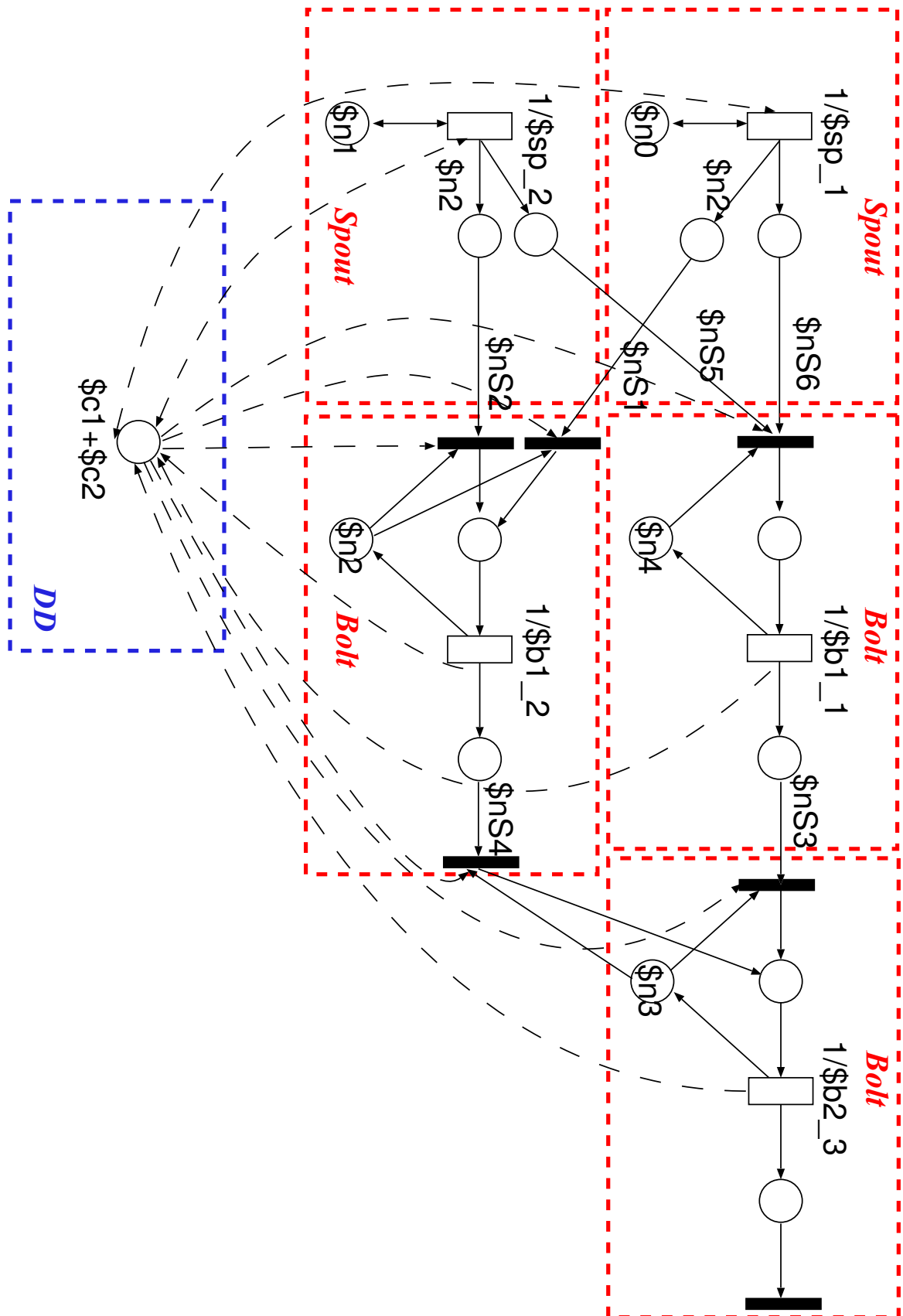


Figure 17: GSPN for the UML activity diagram of Storm

Table 8: Configuration parameters for the first experiment in a single workstation.

NCOMPUTERS	RESMULT	\$SP_I	\$BI_J	PARALLELISM SPOUT_I	PARALLELISM BOLTI_J
1	3	X	100ms	1	1

Table 9: Configuration parameters for the second experiment in a cluster of two workstation.

NCOMPUTERS	RESMULT	\$SP_I	\$BI_J	PARALLELISM SPOUT_I	PARALLELISM BOLTI_J
2	11 (3+8)	X	X	2	2

The performance metric that we have measured in the real system is the average capacity of a bolt. In other words, we have calculated the percentage of time that the threads associated to a bolt are active and working. In the performance model this parameter corresponds to the mean number of tokens in the place *working*. We will have three capacity values per experiment because we have three bolts in the model.

As mentioned above, we have different system configurations that will vary the number of resources and parallelism (tokens in places *spoutParallelism*, *boltParallelism* and *Core*), execution times for bolts (time of processing an input message and generate a new tuple) and spouts (time of inserting a new tuple into the system, i.e., the inverse of the arrival rate). Times are expressed in milliseconds. Two cluster environments have been considered for these tests. We have executed the experiments of the Storm topology 1) locally, in a single workstation (Table 8), and 2) remotely, distributed in two workstations (Table 9). The workstations are Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 32GB of RAM, Gigabit ethernet and Ubuntu Linux version 14.04. Each workstation has 8 cores.

Tables 8 and 9 show the number of cores, the level of parallelism and the execution times of the tasks that will remain constant for the experiments executed locally and remotely. The values of the attribute *numTuples* in Figure 13–17 are  $\$nS1=\$nS2=\$nS5=\$nS6=5$  and  $\$nS3=\$nS4=1$  in both contexts. Column *resMult* tells the total number of available cores in the cluster. The execution of the topology uses 3 cores when it works in a local machine; and 11 cores (3 + 8 cores) when the topology runs remotely. Columns *\$SP\_I* and *\$BI\_J* indicate the execution time of tasks *Spout\_i* and *Bolti\_j* (number of ms per message). By default, *Spout\_1* and *2* are set with the same execution time. We run the experiments by changing the values of execution time for *Spout\_1* and *Spout\_2*. The execution time of *Bolti\_j* remains constant (100 ms/message) for the executions carried out in a single workstation, but it varies when we distribute and execute the topology in the remote cluster. The spouts and bolts have parallelism = 1 (i.e., one thread per component) in Table 8 and parallelism = 2 in Table 9.

On the other hand, we run experiments where the total parallelism of the Storm application was greater than the number of cores. We set parameters of Table 8 as follows: 5 threads (5 components x 1 threads/component) and 3 cores. Table 10 on the next page shows the capacity results obtained by the execution of the topology for the *Bolti\_j* (columns % *BI\_J CAP*), and the deviation with respect to the estimated values obtained by GreatSPN (columns % *BI\_J ERROR*). In some situations, the relative error passes the permissive value of 10% for the bolt capacity. For instance, this happens for the first row of the table: all the bolts process a message every 100 ms and the spouts produce messages with a rate of 0.05 message/ms (20 ms/message). According to the predictions of the performance model (not shown in the table), the thread of *Bolt1\_1* is active 76% of the time, the thread *Bolt1\_2* is active 100% of the time and the thread *Bolt2\_3* is active 35% of the time. Thus, two of the cores are always busy executing the bolts; and the remaining one is devoted to the execution of the spouts (2 threads). Therefore, the workstation is saturated, the OS scheduler rebalances the threads and contention may appear. These facts are potentially the origin of a greater error between the estimated and real performance values.

Finally, we conducted an experiment where the total parallelism of the Storm application was less than the number of cores for the experiment. We set parameters of Table 9 as follows: 10 threads (5 components x 2 threads/component) and 11 cores. In this case, the performance predictions match reasonably good with the performance results of real executions (see results of Table 11). The percentage

Table 10: Validation of the PN when the total number of threads is greater than the number of cores

SP_I	% B1_1 CAP	% B1_2 CAP	% B2_3 CAP	% B1_1 ERROR	% B1_2 ERROR	% B2_3 ERROR
20	35,2	98,7	24,7	116,936	1,317	42,332
30	47,8	92,1	22,8	18,297	8,578	33,692
40	44	88,9	28,3	2,922	2,217	4,296
50	38,4	75,6	23,8	1,666	0,176	4,413
100	22,4	42,2	13,9	11,908	6,077	16,576

Table 11: Validation of the PN when the total number of threads is less or equal than the number of cores

SP_I	B1_1	B1_2	B2_3	% B1_1 CAP	% B1_2 CAP	% B2_3 CAP	% B1_1 ERROR	% B1_2 ERROR	% B2_3 ERROR
20	100	100	100	97,6	100,0	39,0	2,459	9,910	3,716
30	100	100	100	100,0	100,0	43,2	2,913	4,489	6,053
40	100	100	100	100,0	99,5	38,1	1,004	0,407	4,503
50	100	100	100	83,1	78,0	33,1	4,611	2,072	2,426
100	100	100	100	39,4	38,7	17,0	0,068	3,275	4,180
20	20	30	40	41,2	56,4	32,0	2,907	5,852	0,269
30	20	30	40	26,9	40,1	20,8	0,681	0,462	2,868
40	20	30	40	21,2	29,9	16,6	6,057	0,192	4,051
50	20	30	40	16,5	24,7	12,8	3,395	3,216	0,096
100	20	30	40	9,5	11,9	7,3	15,738	0,958	12,618

of relative error is usually less than 10%. The exception is the last row of that table. In this case, the spouts insert tuples at a low speed (100 ms/message or 0.01 message/ms) and bolts are idle most of the time: the capacity is between 7.3 – 11.9 %. The absolute error between the estimations and the real capacity values is small but they are amplified in terms of relative deviation.

From the experiments carried out we get the following insight. If the spouts insert tuples to the system at a low rate (e.g., 100 ms/message or 0.01 message/ms) and the bolts execute low time-consuming functions (e.g., 20 ms/message for Bolt1\_1), the threads will be idle most of the time (e.g., 9, 5% of capacity for Bolt1\_1 in Table 11). Conversely, the threads will be saturated (i.e., capacity 100% for Bolt1\_2 in Table 11) if speed of the spouts inserting messages passes a certain threshold (e.g., less than 20 ms/message or more than 0.05 message/ms) or the bolts execute heavy functions (e.g., more than 100 ms/message for Bolt1\_2).

The work described in this section of the document is the result of the analysis of Apache Storm, which is considered as a reference technology. Most of the concepts can be adapted to other frameworks, since the model is very general. We have shown the transformation of UML activity and deployment diagrams to performance models that capture the essentials of the technology. The experimental results confirm the feasibility of our approach for common Apache Storm configurations.



## 4 Transformations Proposed for the Verification Tool

In deliverable D3.5 [2] we presented the initial version of *D-VerT*, the DICE formal verification tool that allows the designers to verify the design of their applications against safety properties such as reachability of undesired configurations of the system. The formal verification is based on a temporal logic model capturing the behavior of Storm topologies, whose semantics is described in [2]. In this document we focus on the transformations enabling the automated verification of safety properties on Storm applications, starting from DICE-profiled UML diagrams. We first provide an overview of the transformation process implemented in the *D-VerT* tool, then we present the specific UML diagrams adopted to represent Storm topologies and describe the details of the two-step transformation from those diagrams to the formal model. Finally we validate the approach by showing the transformation process applied to one of the DICE industrial use cases.

### 4.1 Overview

Figure 18 depicts the architecture and the execution flow with respect to the different components of *D-VerT*. As described in Sect. 4.2, users define Storm applications as UML Class diagrams on DICE IDE through the Papyrus UML modeling interface. Once the application is completely designed, users specify some tool configuration parameters and launch *D-VerT* tasks. Each task consists of: (i) transformation from the UML model to the temporal logic model; (ii) launch of the verification task; (iii) processing of verification results in order to provide a feedback to the user. The transformation phase is, in its turn, performed in two steps: the first one, detailed in Sect. 4.3 translates the UML Class diagram plus some additional configuration to an intermediate JSON object, and the second one, presented in Sect. 4.3.2 uses the information stored in the JSON object to generate the final Temporal Logic model based on a template file.

### 4.2 DICE Models for Storm Verification

In order to carry out formal verification tasks, Storm topologies have to be specified by means of UML class diagram, conveniently annotated with the DICE::Storm profile. The stereotypes that need to be used are «*StormSpout*», «*StormBolt*» and, optionally, «*StormStreamStep*» (Fig. 19). All of them have already been presented in Sect. 3.2.2. «*StormBolt*» allows the designer to specify parameters as the time needed to process a single tuple (*alpha* tag), the functionality performed by the bolt (*sigma* tag) in terms of the ratio  $\frac{\#output\_tuples}{\#input\_tuples}$ , the minimum and the maximum time to recover from a failure (*minRebootTime* and *maxRebootTime* tags). «*StormSpout*» can be configured by specifying the average emitting rate of the spout (*avg\_emit\_rate* tag). Both «*StormBolt*» and «*StormSpout*» also have the *parallelism* tag, inherited from «*CoreDAGNode*» stereotype.

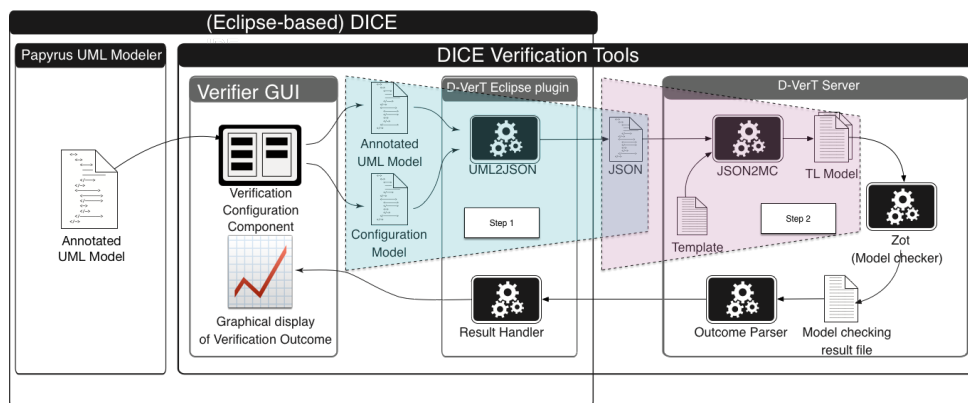


Figure 18: *D-VerT* architecture and execution flow, highlighting the two steps of the transformation from UML Class diagram to the Temporal Logic model.

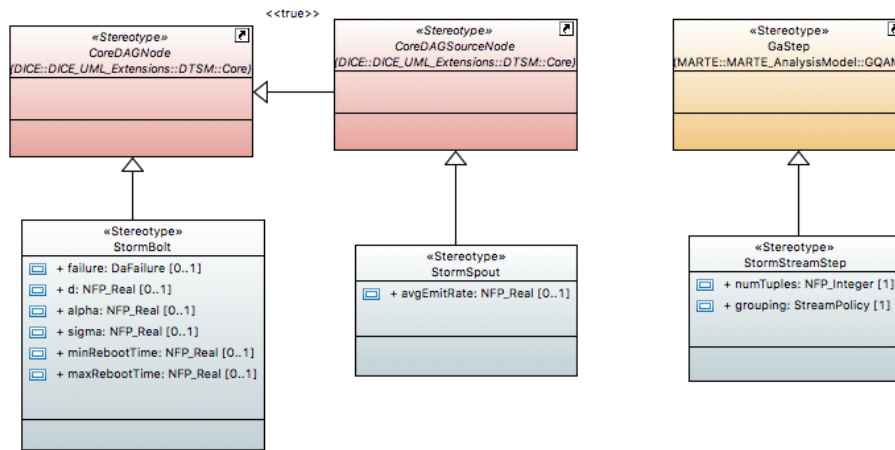


Figure 19: Stereotypes in DICE::Storm profile needed for verification.

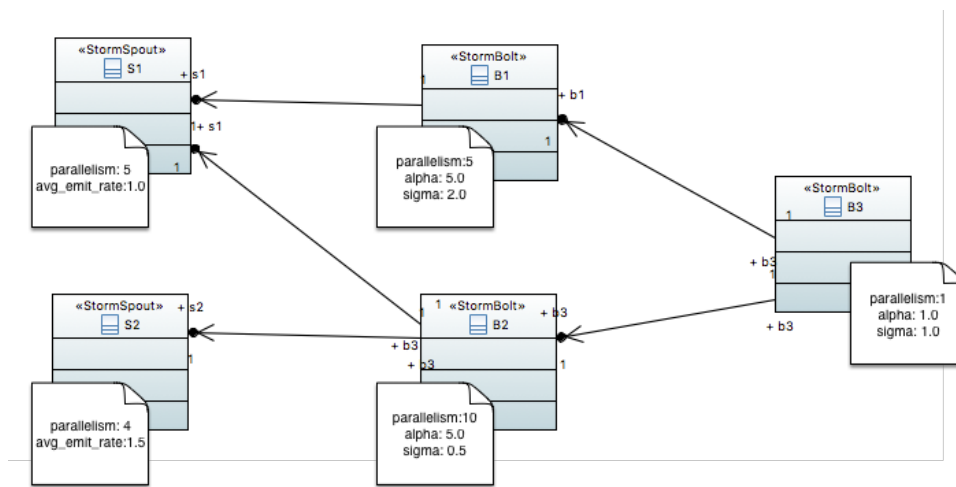


Figure 20: Class diagram representing a simple Storm topology with “direct” subscription. Configuration provided for each node is reported in the annotations.

Spouts and bolts can be instantiated by adding classes to the class diagram and by applying on them the stereotypes *«StormSpout»* and *«StormBolt»*, respectively.

The subscription of a bolt to one or more bolts/spouts can be expressed in two ways. If in the Storm application there are not component that are emitting output to multiple streams, all the subscription can be indicated by directly putting an association from the subscribing bolt to the subscribed component (“direct” way). Otherwise, if a component emits tuples to multiple output streams, each of these streams has to be represented by inserting a class in the diagram and by annotating it as *«StormStreamStep»*. Then the subscribing bolt will have an association to the stream (and not a direct association to the emitting component) and the emitting component will be associated to the stream as well. In this way it is possible to specify explicitly the subscription of bolts to streams (as it is in actual Storm applications). The “direct” way of expressing the subscription is a shortcut provided to the designer to define the topology without further unnecessary details.

Figure 20 shows an example of class diagram representing a topology composed of two spouts (*S1* and *S2*), and three bolts (*B1*, *B2* and *B3*). *«StormSpout»* stereotype is applied on *S1* and *S2*, while *«StormBolt»* is applied on *B1*, *B2* and *B3*. Subscriptions are expressed in the “direct” way: *B1* subscribes to (the output stream of) *S1*, *B2* subscribes to both *S1* and *S2* and *B3* subscribes to both *B1* and *B2*.



### 4.3 Transformations from DICE UML models to formal models

The transformation process is composed of two steps. The first one, performed directly into the DICE IDE by the *UML2Json* component of *D-VerT*, takes a DICE-profiled UML Class diagram as input and produces a JSON object containing all the information needed for the verification task. The second one, performed by the *Json2MC* (standing for “JSON to *Model Checking*”) component, outputs the complete formal model starting from the JSON object. The final result of the transformation is a Common Lisp file containing the temporal logic model, which is then fed to the *Zot*<sup>3</sup> tool. Interested readers can find a detailed description of the temporal logic model in [2].

We decided to split the transformation process in two steps because we wanted to decouple the core verification phase (performed by *Json2MC*) from the UML model definition phase; this allows for greater flexibility when launching the verification phase, as explained below. We defined a JSON schema capturing all the needed information for the verification tasks. This intermediate format allowed us to work on the core verification component independently from the DICE profile definition, and to benefit from its interoperability and openness characteristics. In fact, it is possible to run verification tasks with *Json2MC* also by building the JSON object with other tools (e.g., the architecture recovery tool *OSTIA* presented in [26]).

In the next two sections we provide more details about the two transformation steps.

#### 4.3.1 From DICE UML to JSON

The DICE-profiled UML diagram is produced on the DICE IDE by means of the Papyrus modeling interface. It is saved in XMI format and it is processed by the *UML2Json* component in order to perform the first step of the transformation.

Rather than using a model transformation language, we are using a plain Java component, which navigates the input model and extracts the relevant features to produce the desired output. We adopted this approach, which is analogous to the work performed in [27], to have a lightweight integration in the Eclipse platform, without the need of using any external component.

*UML2Json* is entirely written in Java and relies on the Eclipse *UML2*<sup>4</sup> Java library, an EMF-based implementation of the UML 2.x OMG metamodel for the Eclipse platform.

We defined a set of classes (depicted in Fig. 21) that decorate the elements provided by the Eclipse *UML2* library in order to represent all the needed features included in the DICE-profiled Class diagram for verification.

As shown in Fig. 21 and Listing 2, abstract class *NodeClass* represents generic components of the topology. It decorates the *Class* element provided by *UML2* with additional information such as parallelism level and a textual identifier. *SpoutClass* and *BoltClass* extend *NodeClass* to add the attributes specific to the two kind of components. The *StormTopology* class includes a list of spouts and bolts together with attributes related to the entire topology. *JsonVerificationContext* class encapsulates all the needed information to run verification, combining application-specific information with tool configuration parameters. It includes an attribute of type *StormTopology* and an attribute of type *VerificationParameters*, in which all the needed configuration for the tool can be set. All the data needed for the *StormTopology* objects is gathered by navigating the UML model, while *VerificationParameters* objects are instantiated with data provided by the user by setting up the specific launch configuration on Eclipse. To the extent of this document we will focus on the model navigation process.

Listing 2: Excerpt of the *NodeClass*, representing the generic topology component

```
package it.polimi.dice.verification.uml.diagrams.classdiagram;
import org.eclipse.uml2.uml.Class;
import it.polimi.dice.verification.uml.helpers.UML2ModelHelper;

public abstract class NodeClass {
```

<sup>3</sup><https://github.com/fm-polimi/zot>

<sup>4</sup><http://www.eclipse.org/modeling/mdt/?project=uml2>

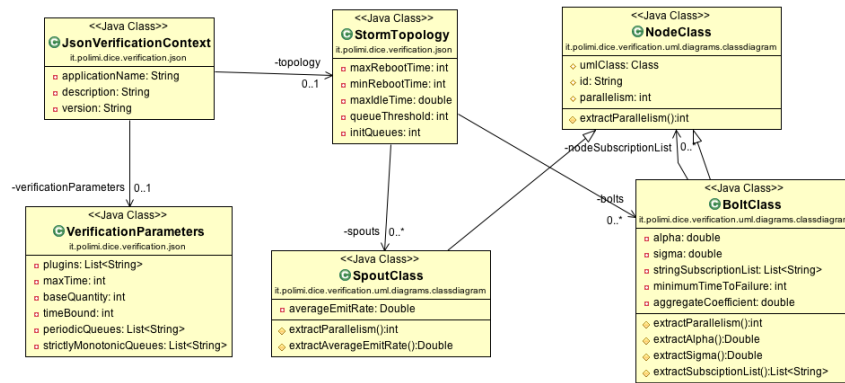


Figure 21: Class Diagram Showing the main Java classes used to enact the transformation.

```

/** The decorated UML2 class element */
protected transient org.eclipse.uml2.uml.Class umlClass;
protected String id;
protected int parallelism;

public NodeClass(org.eclipse.uml2.uml.Class c){
    this.umlClass = c;
    this.id = this.umlClass.getName();
}

/** Extracts parallelism attribute from the class element */
protected abstract int extractParallelism();
...
}

```

The tool extracts from the model all the topology components and instantiates the corresponding Java objects. This is done by simply iterating over the class elements in the UML model and checking if the stereotypes «*StormSpout*» or «*StormBolt*» have been applied on them. When the check is positive, all the attributes are extracted from the model and assigned to the newly created Java object. In the case of bolts, a further exploration of the model is needed to identify the subscription to the streams of other components. As described in Sect. 4.2, bolt subscriptions can be defined in two ways. The simple, "direct" subscription is managed by simply getting all the spout or bolts classes that are directly associated to each bolt. The indirect, stream-specific subscription is processed in two steps: first, the associated stream is identified, then the spout/bolt emitting on the stream is retrieved. An object of the *Topology* class is instantiated as well, and the extracted spouts and bolts are added to it.

Once the Java objects are instantiated, the conversion to JSON objects is straightforward. We used the *gson*<sup>5</sup> Java library to directly perform serialization from one object to the other. The mappings between the class attributes and the JSON object attributes are statically defined in the Java classes by means of the annotation *@SerializedName* when their names are different. Listing 3 shows an example of JSON file produced by *UML2Json* from the UML Class diagram of Fig. 20.

Listing 3: Example JSON file describing a simple topology.

```

1  {
2  "app_name": "SIMPLE-DIA-TOPOLGY",
3  "version": "0.1",
4  "topology": {
5      "spouts": [
6          {
7              "id": "S1",
8              "parallelism": 5,
9              "avg_emit_rate": 1.0,
10             "id": "S2",

```

<sup>5</sup><https://github.com/google/gson>

```

10         "parallelism": 4,
11         "avg_emit_rate":1.5}],
12     "bolts":[
13         {"id":      "B1",
14          "subs":    ["S1"],
15          "alpha":   5.0,
16          "sigma":   2.0,
17          "parallelism": 5},
18         {"id":      "B2",
19          "subs":    ["S1","S2"],
20          "alpha":   5.0,
21          "sigma":   0.5,
22          "parallelism": 10},
23         {"id":      "B3",
24          "subs":    ["B1", "B2"],
25          "alpha":   1.0,
26          "sigma":   1.0,
27          "parallelism": 1}],
28     "min_reboot_time":10,
29     "max_reboot_time":100,
30     "init_queues":0},
31     "verification_params":
32     {"plugin" :["ae2sbvzot"],
33      "max_time" : 20000,
34      "num_steps":15,
35      "periodic_queues":["B1","B2","B3"],
36      "strictly_monotonic_queues":["B1","B2","B3"]}]

```

### 4.3.2 From JSON to the Temporal Logic model

*Json2MC* module exploits the Python templating engine Jinja2<sup>6</sup> to generate, starting from the JSON object and a template file, the temporal logic model. The choice of Jinja2, a popular Python library for the generation of template based-documents, was motivated by its simplicity and lightweight nature. Details about the generation of the temporal logic model can be found in deliverable D3.5 [2]. We recall that the template file is a Common Lisp file with special tags containing variables and expressions that are valued depending on the JSON file. Listing 4 shows the fragment of the template file in which the topology structure is defined: specifically, it is possible to notice the definition of the list of spouts (*the-spouts*), the list of bolts (*the-bolts*) and the hash table (*the-topology-table*) containing, for each bolt, the list of subscribed elements. As shown in Sect. 4.4, the values assigned to those constants and to the hash table are taken from the JSON file. *the-spouts* will be a whitespace-separated list containing the identifier of all the spouts (*topology.spouts* field of the JSON object). In the same way, *the-bolts* will be the list of all of the bolt identifiers, and each of the entries in *the-topology-table* will have a bolt id as key and the list of subscribed elements from that bolt as value.

Listing 4: Template fragment representing the topology configuration.

```

1  ...
2  ;TOPOLOGY DEFINITION
3  (defconstant the-spouts '({{ topology.spouts|join(' ', attribute='id') }}))
4  (defconstant the-bolts '({{ topology.bolts|join(' ', attribute='id') }}))
5
6  {% for b in topology.bolts %}
7  (setf (gethash '{{b.id}} the-topology-table) '({{b.subs | join(' ')}}))
8  {%endfor%}
9  ...

```

<sup>6</sup><http://jinja.pocoo.org/>

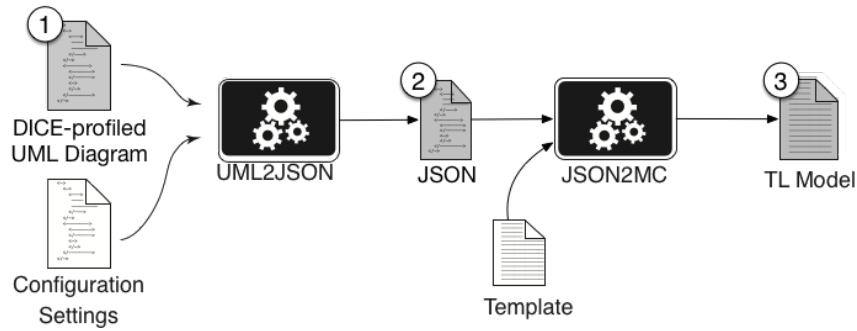


Figure 22: The three stages of the designed application across the transformation process: (1) DICE-profiled UML Class diagram, (2) JSON object and (3) Lisp Temporal Logic model.

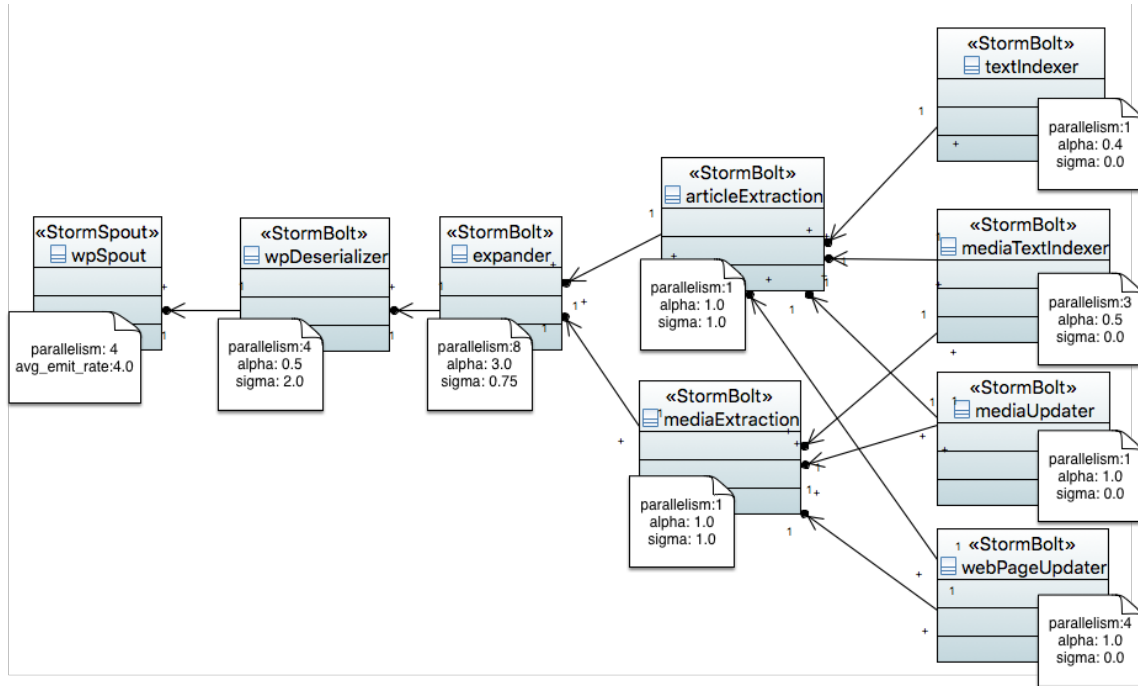


Figure 23: UML Class diagram of *FocusedCrawler* topology.

## 4.4 Validation

In this section we validate the transformations described so far by showing an example of the whole transformation process from DICE-profiled UML Class diagram to the temporal logic model. We remark that the goal of this section is not to validate the *D-VerT* and its associated verification workflow (this validation has already been performed in deliverable D3.5 [2]), but only the transformation steps that produce the formal model from the DICE-profiled UML model.

Figure 22 summarizes the transformation process and highlights the three stages of the model that are shown below in this section: the DICE-Profiled UML Class diagram (Fig. 23), the JSON object (Listing 5) and the Lisp Temporal Logic model (Listing 6).

The example topology is the *FocusedCrawler* topology, taken from the use case of one of the industrial partners<sup>7</sup>. Figure 23 shows the UML design of the topology. Designer have to provide values to all the relevant parameters for verification for each component.

Those values are read, together with the topology structure, by *UML2Json*, which generate the JSON file shown in Listing 5.

<sup>7</sup><https://github.com/socialsensor/storm-focused-crawler/blob/master/src/main/java/eu/socialsensor/focused/crawler/FocusedCrawler.java>

Listing 5: JSON File produced by *UML2Json*.

```

1 {
2   "app_name": "FOCUSED-CRAWLER-COMPLETE-ALL",
3   "description": "",
4   "version": "0.1",
5   "topology": {
6     "bolts": [
7       {"id": "WpDeserializer",
8        "parallelism": 4,
9        "subs": ["wpSpout"],
10       "alpha": 0.5,
11       "sigma": 2.0},
12      {"id": "expander",
13       "parallelism": 8,
14       "subs": ["WpDeserializer"],
15       "alpha": 3.0,
16       "sigma": 0.75},
17      {"id": "articleExtraction",
18       "parallelism": 1,
19       "subs": ["expander"],
20       "alpha": 1.0,
21       "sigma": 1.0},
22      {"id": "mediaExtraction",
23       "parallelism": 1,
24       "subs": ["expander" ],
25       "alpha": 1.0,
26       "sigma": 1.0},
27      {"id": "webPageUpdater",
28       "parallelism": 4,
29       "subs": ["articleExtraction", "mediaExtraction" ],
30       "alpha": 1.0,
31       "sigma": 1.0},
32      {"id": "textIndexer",
33       "parallelism": 1,
34       "subs": ["articleExtraction"],
35       "alpha": 0.4,
36       "sigma": 0.0},
37      {"id": "mediaupdater",
38       "parallelism": 1,
39       "subs": ["articleExtraction", "mediaExtraction"],
40       "alpha": 1.0,
41       "sigma": 0.0},
42      {"id": "mediatextindexer",
43       "parallelism": 3,
44       "subs": ["articleExtraction", "mediaExtraction"],
45       "alpha": 0.5,
46       "sigma": 0.0}
47     ],
48     "spouts": [
49       {"id": "wpSpout",
50        "avg_emit_rate": 4.0,
51        "parallelism": 4}
52     ],
53     "min_reboot_time": 10,
54     "max_reboot_time": 100,
55     "max_idle_time": 1.0,
56     "init_queues": 4
57   },
58   "verification_params":

```

```

59     {"plugin" :["ae2bvzot", "ae2sbvzot"],
60      "max_time" : 20000,
61      "num_steps":20,
62      "periodic_queues":["WpDeserializer", "expander",
        "articleExtraction","mediaExtraction", "webPageUpdater","textIndexer",
        "mediaupdater", "mediatextindexer"]}
63 }

```

As final step in the transformation process, *Json2MC* takes the JSON object and generates the final Lisp file based on its content. Listing 6 shows how the fragment of template presented in Listing 4 is rendered according to the JSON object of Listing 5. It can be noticed that the values of `the-spouts` and `the-bolts` constants correspond to the identifiers of the spouts and bolts present first in the UML diagram and then in the JSON object. In the same way., `the-topology-table` hash table is populated with values that reflect the subscription relationship between components defined in the starting model.

Listing 6: Excerpt of the Lisp file produced by *Json2MC*.

```

1  ...
2  ;TOPOLOGY DEFINITION
3  (defconstant the-spouts '(wpSpout))
4  (defconstant the-bolts '(WpDeserializer expander articleExtraction mediaExtraction
        webPageUpdater textIndexer mediaupdater mediatextindexer))
5
6  ;hash table containing the subscription lists of all the bolts
7  (defvar the-topology-table)
8  (setq the-topology-table (make-hash-table :test 'equalp))
9
10 (setf (gethash 'WpDeserializer the-topology-table) '(wpSpout))
11 (setf (gethash 'expander the-topology-table) '(WpDeserializer))
12 (setf (gethash 'articleExtraction the-topology-table) '(expander))
13 (setf (gethash 'mediaExtraction the-topology-table) '(expander))
14 (setf (gethash 'webPageUpdater the-topology-table) '(articleExtraction
        mediaExtraction))
15 (setf (gethash 'textIndexer the-topology-table) '(articleExtraction))
16 (setf (gethash 'mediaupdater the-topology-table) '(articleExtraction mediaExtraction))
17 (setf (gethash 'mediatextindexer the-topology-table) '(articleExtraction
        mediaExtraction))
18 ...

```

## 5 Conclusions

In this document we have presented the transformation of UML profiled models at DPIM and DTSM levels into quality analysis models for studying (a) the performance and reliability of software systems at design level; and (b) the verification of safety requirements using formal verification techniques. The performance models obtained by the transformation of the UML diagrams are used in the DICE Simulation Tool; and the formal models obtained for the verification of safety properties are used in the DICE Verification Tool.

In particular, for the transformation to performance models, we have considered UML profiled diagrams at DPIM level; and UML profiled diagrams for Hadoop MapReduce and Storm technologies at DTSM level. For the transformation to formal verification models, we have considered UML profiled diagrams for the Storm technology at DTSM level.

Table 12 summarizes the main achievements of this deliverable in terms of compliance with the initial set of requirements presented in Section 2. In Table 12, 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).

The level of fulfillment of the requirements is determined according to several aspects. The criteria for determining the level of fulfillment for the requirements of the transformations are:

- The existence of a theoretical transformation of the UML models annotated with the DICE profile into a performance or formal verification model; i.e., the existence of a conceptual mapping between elements of the UML profiled diagrams and the destination model.
- The validation of the theoretical transformation by comparing the results obtained from the evaluation of the transformed models in the simulation and verification tools with respect to the experimental results extracted from the real execution of the application on a controlled environment.
- The implementation of the theoretical transformations in the transformation tool.

Table 12: Level of compliance of the current version with the initial set of requirements

REQUIREMENT	TITLE	PRIORITY	LEVEL OF FULFILLMENT
R3.1	M2M Transformation	Must have	✓
R3.2	Taking into account relevant annotations	Must have	✓
R3.3	Transformation rules	Could have	✓
R3.6	Transparency of underlying tools	Must have	✓
R3.7	Generation of traces from the system model	Must have	✓
R3.12	Modelling abstraction level	Must have	✓
R3.13	White/black box transparency	Must have	✓
R3.15	Verification of temporal safety/privacy properties	Must have	✓
R3IDE.3	Usability	Could have	✗



According to these considerations, the requirement R3.1 is partially-high supported. On the one hand, the transformation of the annotated DPIM models (i.e., UML activity and sequence diagrams) into performance models for the DICE Simulation Tool is completely supported. On the other hand, the transformation of the annotated DTSM diagrams into performance models for the DICE Simulation Tool is currently under implementation for Hadoop MapReduce and Storm technologies. Nevertheless, we have identified the technological concepts, defined the theoretical transformation and validated the approach for those technologies. The transformation of the annotated UML models into formal models for the DICE Verification Tool is fully supported for the Storm technology at the DTSM level.

The rest of the requirements of Table 12 are related to the specific features of the transformation tools (e.g., R3.6, transparency to the users); and the integration of the transformations with the DICE Simulation and Verification tools. Most of the requirements are supported or partially-high supported. In the Future Work section, we will list the actions for accomplishing the unsupported and partially supported requirements.

## 5.1 Further Work

Task 3.1 will not produce more additional deliverables. The remaining requirements and features of Table 12 will be carried out in the following months and they will be reported through alternative channels such as publications in conference and journal papers.

In general, we will finish the implementation of the transformations of Hadoop MapReduce and Storm at DTSM level to performance models. We expect to develop new transformations of UML diagrams annotated at DTSM level to performance and formal verification models for other Big Data technologies such as, for example, Tez or Spark.

In particular, we list the concrete actions for each partially or unsupported requirement:

- Regarding requirement R3.1, the definition and implementation of more transformations for Big Data technologies will be addressed by the simulation and verification tool.
- Regarding requirement R3.2, we need to address annotations for reliability models.
- Regarding requirement R3.15, more safety properties may be addressed. Privacy properties are currently considered and we plan to address them in the next months.
- Regarding requirement R3IDE.3, this requirement needs to be updated since it will apply only to the simulation tool.



## References

- [1] The DICE Consortium. *DICE Simulation Tool - Initial Version*. Tech. rep. URL: [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D3.2\\_DICE-simulation-tools-Initial-version.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D3.2_DICE-simulation-tools-Initial-version.pdf). European Union's Horizon 2020 research and innovation programme, 2016.
- [2] The DICE Consortium. *DICE Verification Tool - Initial Version*. Tech. rep. URL: [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D3.5\\_DICE-verification-tools-Initial-version.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/02/D3.5_DICE-verification-tools-Initial-version.pdf). European Union's Horizon 2020 research and innovation programme, 2016.
- [3] The DICE Consortium. *Design and Quality abstractions - Initial Version*. Tech. rep. URL: <https://vm-project-dice.doc.ic.ac.uk/redmine/projects/dice/repository/show/WP2/D2.1/submitted/D2.1.pdf>. European Union's Horizon 2020 research and innovation programme, 2015.
- [4] The DICE Consortium. *DICE Model Repository*s. URL: <https://github.com/dice-project/DICE-Models>. Dec., 2015.
- [5] The DICE Consortium. *DICE Profiles Repository*. URL: <https://github.com/dice-project/DICE-Profiles>. Dec., 2015.
- [6] The DICE Consortium. *DICE Simulation Repository*. URL: <https://github.com/dice-project/DICE-Simulation>. Dec., 2015.
- [7] The DICE Consortium. *DICE Verification Repository*. URL: <https://github.com/dice-project/DICE-Verification>. Dec., 2015.
- [8] The DICE Consortium. *Requirement Specification*. Tech. rep. URL: [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2\\_Requirement-specification.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2_Requirement-specification.pdf). European Union's Horizon 2020 research and innovation programme, 2015.
- [9] The DICE Consortium. *Requirement Specification - Companion Document*. Tech. rep. URL: [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2\\_Requirement-specification\\_Companion.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.2_Requirement-specification_Companion.pdf). European Union's Horizon 2020 research and innovation programme, 2015.
- [10] *DICE Requirement List (Online version)*. 2015. URL: [https://docs.google.com/spreadsheets/d/1Wn90XGsTknrAs5ASUadOp9IpQ9BM\\_WM4NsyAuXL6\\_Ug/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1Wn90XGsTknrAs5ASUadOp9IpQ9BM_WM4NsyAuXL6_Ug/edit?usp=sharing).
- [11] The Eclipse Foundation & Obeo. *Acceleo*. URL: <https://eclipse.org/acceleo/>. Dec., 2015.
- [12] OMG. *MOF Model to Text Transformation Language (MOFM2T), 1.0*. URL: <http://www.omg.org/spec/MOFM2T/1.0/>. Object Management Group, Jan. 2008. URL: <http://www.omg.org/spec/MOFM2T/1.0/>.
- [13] Dipartimento di informatica, Università di Torino. *GGraphical Editor and Analyzer for Timed and Stochastic Petri Nets*. URL: [www.di.unito.it/~greatspn/index.html](http://www.di.unito.it/~greatspn/index.html). Dec., 2015.
- [14] The DICE Consortium. *Demonstrators Implementation Plan*. Tech. rep. URL: <http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/06/D6.1-Demonstrators-implementation-plan.pdf>. European Union's Horizon 2020 research and innovation programme, 2016.
- [15] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems, Version 1.1*. URL: <http://www.omg.org/spec/MARTE/1.1/>. Object Management Group, June 2011. URL: <http://www.omg.org/spec/MARTE/1.1/>.
- [16] The DICE Consortium. *Transformations to Analysis Models — Companion Document*. Tech. rep. URL: <http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/07/D3.1-Companion.pdf>. European Union's Horizon 2020 research and innovation programme, 2016.

- [17] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*. URL: <http://www.omg.org/spec/QVT/1.1/>. Object Management Group, Jan. 2011. URL: <http://www.omg.org/spec/QVT/1.1/>.
- [18] Simona Bernardi et al. “A Systematic Approach for Performance Evaluation Using Process Mining: The POSIDONIA Operations Case Study”. In: *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*. QUDOS 2016. Saarbrücken, Germany: ACM, 2016, pp. 24–29. ISBN: 978-1-4503-4411-1. DOI: 10.1145/2945408.2945413. URL: <http://doi.acm.org/10.1145/2945408.2945413>.
- [19] B. F. van Dongen et al. “The Prom Framework: A New Era in Process Mining Tool Support”. In: *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets*. ICATPN’05. Miami: Springer-Verlag, 2005, pp. 444–454. ISBN: 3-540-26301-2, 978-3-540-26301-2. DOI: 10.1007/11494744\_25. URL: [http://dx.doi.org/10.1007/11494744\\_25](http://dx.doi.org/10.1007/11494744_25).
- [20] Simona Bernardi, José Merseguer, and Dorina C. Petriu. “A dependability profile within MARTE”. In: *Software & Systems Modeling* 10.3 (2011), pp. 313–336. ISSN: 1619-1374. DOI: 10.1007/s10270-009-0128-1. URL: <http://dx.doi.org/10.1007/s10270-009-0128-1>.
- [21] Giovanni Chiola et al. “Stochastic well-formed colored nets and symmetric modeling applications”. In: *IEEE Transactions on Computers* 42.11 (1993), pp. 1343–1360.
- [22] The DICE Consortium. *Optimization Tools - Initial Version*. Tech. rep. European Union’s Horizon 2020 research and innovation programme, 2016.
- [23] Danilo Ardagna et al. “Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets”. Paper submitted for publication on the 16th International Conference on Algorithms and Architectures for Parallel Processing. 2016.
- [24] *Apache Hive*. URL: <https://hive.apache.org> (visited on 03/09/2016).
- [25] Giovanni Chiola et al. “Generalized stochastic Petri nets: A definition at the net level and its implications”. In: *IEEE Transactions on software engineering* 19.2 (1993), pp. 89–107.
- [26] Marcello M. Bersani et al. “Continuous Architecting of Stream-Based Systems”. In: *13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Venice, Italy, April 5-8, 2016*. 2016, pp. 146–151. DOI: 10.1109/WICSA.2016.26. URL: <http://dx.doi.org/10.1109/WICSA.2016.26>.
- [27] Alfredo Motta. “Logic-based verification of multi-diagram UML models for timed systems”. In: (2013).
- [28] Stuart Kent. “Model Driven Engineering”. In: *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*. Ed. by Michael J. Butler, Luigia Petre, and Kaisa Sere. Vol. 2335. Lecture Notes in Computer Science. Springer, 2002, pp. 286–298. ISBN: 3-540-43703-7.
- [29] OMG. *Common Object Request Broker Architecture: Core Specification*. Object Management Group, Mar. 2004. URL: <http://www.omg.org/spec/CORBA/3.0.3/>.
- [30] OMG. *Object Management Group*. 2011. URL: <http://www.omg.org>.
- [31] OMG. *MDA Guide Version 1.0.1*. Object Management Group, June 2003. URL: <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [32] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group, Jan. 2006. URL: <http://www.omg.org/spec/MOF/2.0/>.
- [33] *Unified Modeling Language: Infrastructure*. Version 2.4.1, OMG document: formal/2011-08-05. 2011.
- [34] OMG. *OCL 2.2 Specification*. Object Management Group, Feb. 2010. URL: <http://www.omg.org/spec/OCL/2.2/>.

- [35] Abel Gómez et al. “Towards a UML Profile for Data Intensive Applications”. In: *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*. QUDOS 2016. Saarbrücken, Germany: ACM, 2016, pp. 18–23. ISBN: 978-1-4503-4411-1. DOI: 10.1145/2945408.2945412. URL: <http://doi.acm.org/10.1145/2945408.2945412>.
- [36] Tadao Murata. “Petri Nets: Properties, Analysis and Applications.” In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580.
- [37] ISO. *Systems and software engineering – High-level Petri nets – Part 2: Transfer format*. ISO/IEC 15909-2:2011. Geneva, Switzerland, 2008.
- [38] Lom Messan Hillah et al. “PNML Framework: an extendable reference implementation of the Petri Net Markup Language”. In: *31st International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2010)*. Vol. 6128. Lecture Notes in Computer Science. MoVe INT LIP6. Braga, Portugal: Springer, June 2010, pp. 318–327.
- [39] *Apache Hadoop*. <http://hadoop.apache.org>. URL: <http://hadoop.apache.org> (visited on 11/17/2015).
- [40] *Apache Storm Website*. URL: <http://storm.apache.org/>.

## A Background

The term Model-Driven Engineering (MDE) was proposed by Kent [28] as a general framework to carry out a software development. MDE aims at organizing software artifacts at different abstraction levels, advocating for the use of models as the key artifacts to be built and maintained. A model consists of a set of elements that provide a precise and abstract description of a system from a view point.

In this general framework, the software development process becomes thus a series of refinements and/or transformations of models where the abstraction level changes on each step (e.g., models become closer to the implementation platform). An MDE process must clearly define the sequence of models to develop at each level and must describe how to refine models in order to decrease the level of abstraction.

To address the standardisation issues, the Object Management Group (OMG)<sup>8</sup> [30] launched the Model-Driven Architecture (MDA) initiative [31] as an approach to specify interoperable systems by using formal (or semi-formal) models. The DICE framework heavily relies on the grounds set out by MDA, and as such, on several other OMG standards such as MOF (Meta Object Facility [32]), UML (Unified modeling Language [33]), OCL (Object Constraint Language [34]), MARTE (Modeling and Analysis of Real Time and Embedded systems [15]) and QVT (MOF 2.0 Query/View/Transformation [17]).

Based on the concepts provided by the previous standards, it is possible to define MDE processes. A model transformation can be considered a trivial MDE process. Figure 24 depicts the main elements playing a role in a model transformation in a schematic way. The diagram describes the artifacts involved in a model transformation, i.e., two candidate models (and metamodels) only. In the example, a set of *Rules* defines how to transform concepts from the metamodel *MetaModel 1* to the metamodel *MetaModel 2*. Thus, by applying the rules, an initial model (*Model 1*, which conforms to *MetaModel 1*) is automatically transformed to obtain the *Model 2*, which conforms to *MetaModel 2*.

As it can be observed, model transformations are described using the base concepts defined in their metamodels (the so called metaclasses), and subsequently, this is how the DICE transformations have been defined. Next subsections, first describe the main metamodels and domain concepts that play an important role in the transformations to the analysis models (Subsections A.1 and A.2); and second introduce the basic concepts of the Big Data technologies that we will consider for the transformations of the DTSM to performance and formal verification models (Subsection A.3).

### A.1 Unified Modeling Language

The Unified Modeling Language (UML) [33] is a standard which provides a language to describe different systems. UML is a domain-independent language, although its origins are in the object-oriented modeling. A UML model consists of elements such as packages, classes, and associations. The corresponding UML diagrams are graphical representations of parts of the UML model. These diagrams contain graphical elements (nodes connected by paths) that represent elements in the UML model.

UML modeling can be generally divided into two semantic categories: behavioral modeling and structural modeling. In addition to these main categories, there are some supplemental modeling con-

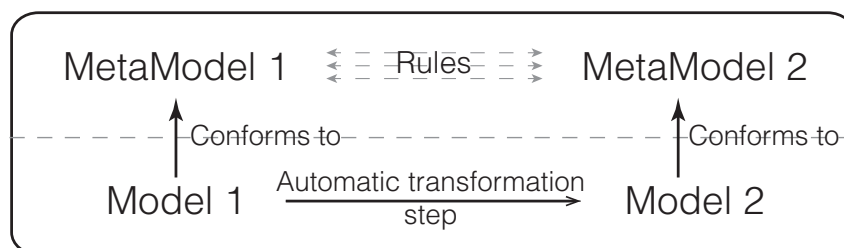


Figure 24: Example of a model transformation

<sup>8</sup>The OMG is a consortium founded in 1989 aimed at setting modeling and object-oriented standards. The OMG released their first standard, CORBA [29], in 1991. Since then, several specifications which can be considered as a *de facto* standards in industry are promoted by them, (e.g. UML or MOF).

structs that have both behavioral and structural aspects (e.g., deployment diagrams, use cases) which may complement behavioral and/or structural models. This section will focus in the former category, i.e., behavioral modeling. Specifically, we will focus on *activity diagrams* and *sequence diagrams*, which are the two kind of diagrams that will be transformed to quality analysis models.

## Activity Diagrams

An *Activity*, as specified in the UML standard [33], is a kind of *Behavior* that is specified as a graph of *nodes* interconnected by *edges*. A subset of the nodes are *executable nodes* that embody lower-level steps in the overall Activity. *Object nodes* hold data that is input to and output from executable nodes, and moves across *object flow edges*. *Control nodes* specify sequencing of executable nodes via *control flow edges*. Activities are essentially what are commonly called “control and data flow” models.

Figure 25 shows the main concepts (i.e., metaclasses) used to create activity diagrams together with their relationships. The most relevant elements from the transformation to analysis models point of view are:

**UML::Packages::Model** — A Model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose.

**UML::Activities::Activity** — An Activity is the specification of parameterized behavior as the coordinated sequencing of subordinate units (ActivityNodes).

**UML::Activities::ActivityNode** (abstract) — ActivityNode is an abstract class for points in the flow of an Activity connected by ActivityEdges.

**UML::Activities::ActivityEdge** (abstract) — An ActivityEdge is an abstract class for directed connections (i.e., with source [1..1] and target [1..1]) between two ActivityNodes.

**UML::Activities::ControlFlow** — A ControlFlow is an ActivityEdge traversed by control tokens or object tokens of control type, which are used to control the execution of ExecutableNodes.

**UML::Activities::ControlNode** (abstract) — A ControlNode is an abstract ActivityNode that coordinates flows in an Activity.

**UML::Activities::InitialNode** — An InitialNode is a ControlNode that offers a single control token when initially enabled.

**UML::Activities::FinalNode** (abstract) — A FinalNode is an abstract ControlNode at which a flow in an Activity stops.

**UML::Activities::ActivityFinalNode** — An ActivityFinalNode is a FinalNode that terminates the execution of its owning Activity.

**UML::Activities::ForkNode** — A ForkNode is a ControlNode that splits a flow into multiple concurrent flows.

**UML::Activities::JoinNode** — A JoinNode is a ControlNode that synchronizes multiple flows.

**UML::Activities::DecisionNode** — DecisionNode is a ControlNode that chooses between outgoing ActivityEdges for the routing of tokens.

**UML::Activities::MergeNode** — A MergeNode is a ControlNode that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.

**UML::Actions::Action** (abstract) — An Action is the fundamental unit of executable functionality. The execution of an Action represents some transformation or processing in the modeled system. Actions provide the ExecutableNodes within Activities and may also be used within Interactions.

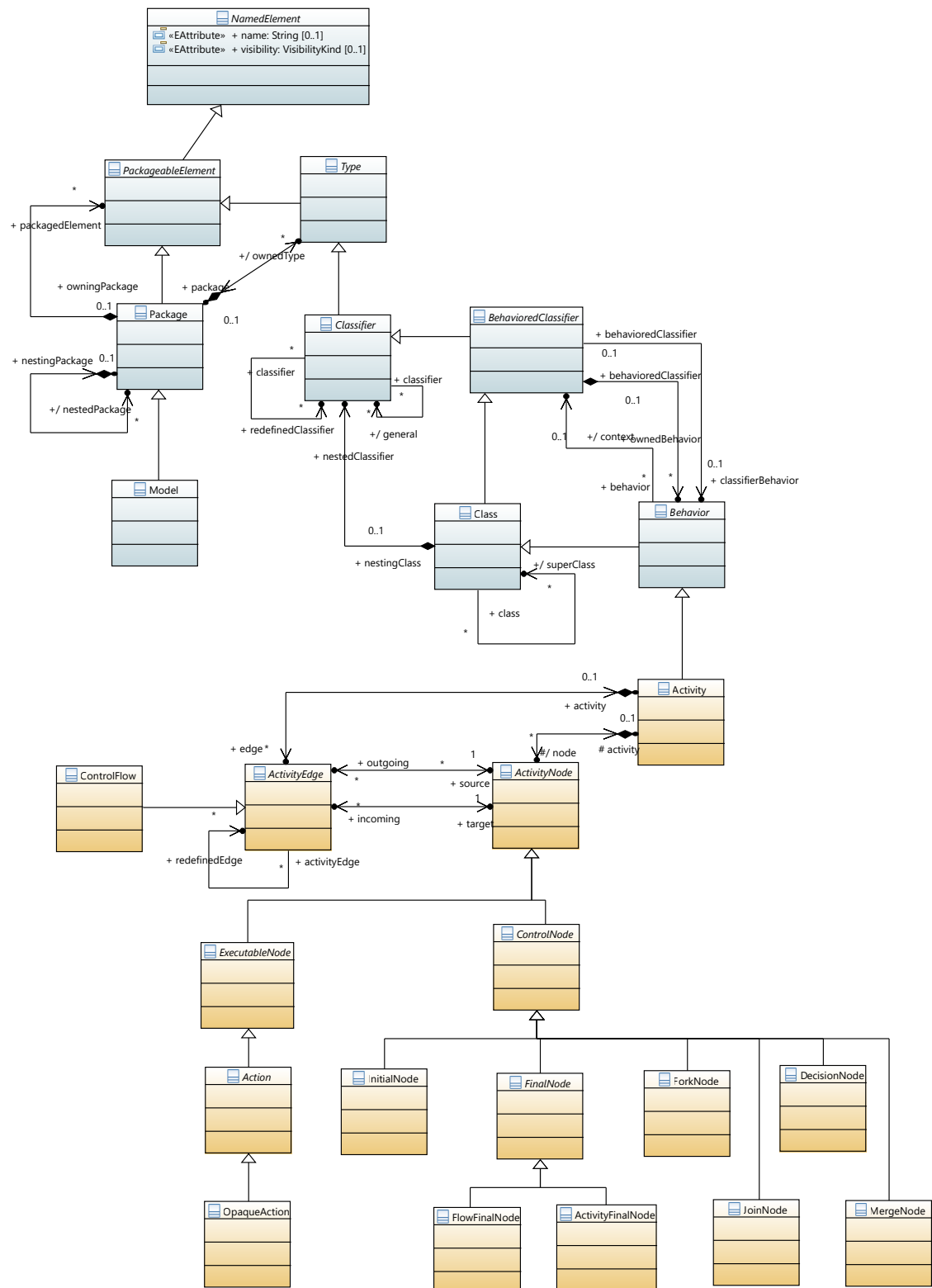


Figure 25: Metaclasses of the UML Activity Diagram



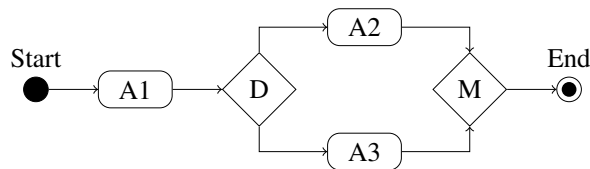


Figure 26: Sample Activity Diagram

**UML::Actions::OpaqueAction** — An *OpaqueAction* is an *Action* whose functionality is not specified within UML.

Figure 26 shows an example activity diagram using the standard notation. It is composed by an *InitialNode* (Start), an *ActivityFinalNode* (End), three *OpaqueActions* (A1, A2 and A3), a *DecisionNode* (D) and a *MergeNode* (M).

## Sequence Diagrams

*Interactions* are presented in the UML standard [33] as a mechanism to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. The most visible aspects of an *Interaction* are the *messages* between *lifelines*. The sequence of the messages is considered important for the understanding of the situation. The data that the messages convey and the lifelines store may also be very important, but the *Interactions* do not focus on the manipulation of data even though data can be used to decorate the diagrams.

Figure 27 shows the UML metaclasses that may be used to create activity diagrams. The most relevant elements from the transformation to analysis models point of view are:

**UML::Packages::Model** — As aforementioned a *Model* captures a view of a system with a certain purpose.

**UML::Interactions::Interaction** — *Interactions* are units of behavior, and focus on the passing of information with *Messages* between the *ConnectableElements*. *Interactions* are the top-level elements of a UML sequence diagram.

**UML::Interaction::Lifeline** — A *Lifeline* represents an individual participant in the *Interaction*. *Lifelines* represent only one interacting entity (via the *represents* association to *ConnectableElement*).

**UML::Interactions::InteractionFragment** (abstract) — *InteractionFragment* is an abstract notion of the most general interaction unit. An *InteractionFragment* is a piece of an *Interaction*. Each *InteractionFragment* is conceptually like an *Interaction* by itself. *InteractionFragments* are the main constituent parts of an *Interaction*. An *InteractionFragment* may either be contained directly in an enclosing *Interaction*, or may be contained within an *InteractionOperand* of a *CombinedFragment*. As a *CombinedFragment* is itself an *InteractionFragment*, there may be multiple nesting levels of *InteractionFragments* within an *Interaction*.

**UML::Interactions::CombinedFragment** — The semantics of a *CombinedFragment* is dependent upon the contained *InteractionOperand*. Typical operators are *alt* (for alternative behaviors), *opt* (for optional behaviors) or *par* (for parallel execution paths).

**UML::Interactions::InteractionOperan** — As previously introduced, an *InteractionOperand* is a region within a *CombinedFragment*.



Figure 27: Metaclasses of the UML Sequence Diagram



**UML::Interactions::ExecutionSpecification** (abstract) — An *ExecutionSpecification* is a specification of the execution of a unit of Behavior or Action within the Lifeline. The duration of an *ExecutionSpecification* is represented by two *OccurrenceSpecification*, the start *OccurrenceSpecification* and the finish *OccurrenceSpecification*.

**UML::Interactions::BehaviorExecutionSpecification** — An *ExecutionSpecification* linked to a Behavior.

**UML::Interactions::ActionExecutionSpecification** — An *ExecutionSpecification* linked to an Action.

**UML::Interactions::OccurrenceSpecification** — An *OccurrenceSpecification* is the basic semantic unit of Interactions. The sequences of occurrences specified by them are the meanings of Interactions.

**UML::Interactions::ExecutionOccurrenceSpecification** — An *ExecutionOccurrenceSpecification* represents moments in time at which Actions or Behaviors start or finish.

**UML::Interactions::MessageOccurrenceSpecification** — A *MessageOccurrenceSpecification* specifies the occurrence of Message events. A *MessageOccurrenceSpecification* is a kind of *MessageEnd*. Messages are generated either by synchronous Operation calls or asynchronous Signal sends.

**UML::Interactions::MessageEnd** — *MessageEnd* is an abstract specialization of *NamedElement* that represents what can occur at the end of a Message.

**UML::Interactions::Message** — A Message is simply the trace between a message *send event* and a *receive event*.

Figure 28 shows an example sequence diagram using the standard notation. It describes the interaction of a user with a two tier application. The figure contains three Lifelines represented by dotted lines: one for *user* (which is an instance of the *User* classifier), a second one for *ui* (which represents and instance of *UI*) and a third one for *backend* (which is an instance of the *Backend* classifier). The three white boxes drawn on top the *lifelines* represent *ExecutionSpecifications* (either *Action-ExecutionSpecifications* or *BehaviorExecutionSpecifications*). The arrow labeled with *displayReport(id)* represents a message from *user* to *ui* requesting the operation *displayReport* defined by the UI classifier. The operation takes as an argument the *id* value. Similarly, the arrow labeled with *getReport* represents a message from *ui* to *backend*. Finally, the dashed arrows labeled with *report*, represent the reply messages from the *backend* to the *ui*, and from the *ui* to the *user*.

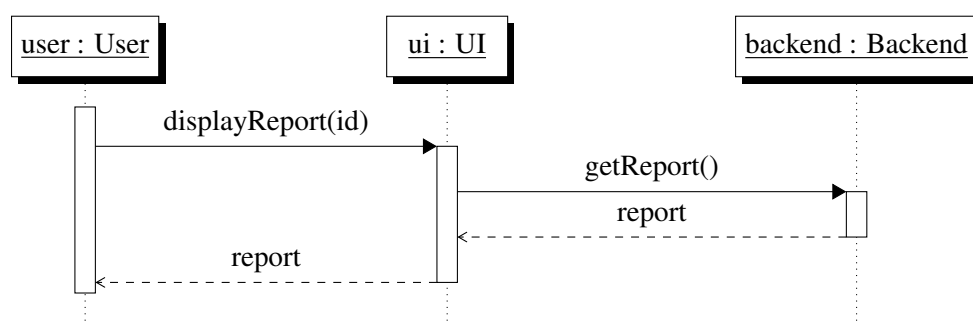


Figure 28: Sample Sequence Diagram

## The DICE Profile, Non Functional Properties (NFP) and the Value Specification Language (VSL)

The DICE Profile [3, 35] is an extension mechanism that allows converting the Unified Modeling Language into a domain specific modeling language for quality evaluation of data-intensive applications. For quality assessment, the DICE profile relies on two already existing UML profiles, namely the standard MARTE profile [15] and the DAM profile (*Dependability Analysis and Modeling*) [20]. From MARTE, DICE applies the *NFPs* and *VSL* sub-profiles, while from DAM it imports the *DAM Library* which also imports the *MARTE Library*. The MARTE NFP modeling framework [15, Chapter 8] provides the capability to describe various kind of values related to physical quantities (e.g., Time, Frequency, Energy...). These values are used to describe the non-functional properties of a system. As aforementioned, DICE makes use of the MARTE VSL subprofile, which provides a textual language for specifying the values of constraints, properties, and stereotype attributes, particularly related to *Non Functional Properties* (NFP). The VSL is a complex language that, in fact, can be used by profile users in tagged values, body of constraints, and in any UML element associated with value specifications. A complete reference of the MARTE NFP types can be found in Appendix B, and a complete description of the VSL language can be found in Annex B of the MARTE standard [15].

VSL expressions are used in DICE-profiled models with two main goals: (i) to specify the values of NFP that must be used during the simulation (i.e., to specify input data) and (ii) to specify the NFP that must be calculated by a given simulation (i.e., to specify the output results). An example VSL expression for a tagged value *t* of type *NFP\_Frequency* is:

$$\begin{array}{cccc} (\text{expr}=2*\$freq, & \text{unit}=\text{Hz}, & \text{statQ}=\text{mean}, & \text{source}=\text{est}) \\ (1) & (2) & (3) & (4) \end{array}$$

This expression specifies that, when analyzing the anotated model containing the tagged value *t*, *t* will be a frequency specified in *Hertz* (2), whose mean value (3) will be twice the value of the variable *\$freq* (1). That value will be obtained from an estimation (4).

## A.2 Petri Net Modeling

Petri nets [36] are one of the mathematical formalisms chosen to specify the analysis models in the Simulation Tool. Basic Petri nets (also known as *Place/Transition Nets*) are graphically represented as a directed graph composed by *places*, *transitions* and *arcs*. *Arcs* run from a place to a transition or vice versa. *Places* may be marked by a set of *tokens*. Finally, *transitions* may be *fired* when all its preceding places have enough tokens. When a transition is fired, it consumes tokens in all the preceding places and creates tokens in the following places.

Petri nets are well suited for modeling concurrent systems, and some extensions to the basic *Place/Transition Nets* even allow to model, analyse, and evaluate quality properties of the systems under study in a quantitative way. In this section, first we focus on the particular type of Petri nets that are finally used as performance models for the simulation; and second, we present the Petri Net Markup Language (PNML), an intermediate format for representing Petri nets that is used during the transformation process.

### A.2.1 Colored and Stochastic Petri Nets

Among all the possible variants of Petri nets in the literature, two of them are suitable for the simulation of UML diagrams annotated with the DICE profile for the Big Data technologies under consideration: *Generalized Stochastic Petri Nets* and *Stochastic Well-formed colored Nets*. A *Generalized Stochastic Petri Net* (GSPN) [25] is a Petri net extended with a temporal interpretation. A GSPN has places and transitions like *Place/Transition Nets*. In this case, places represent the intermediate steps of the processing and transitions are fired when certain conditions are met; or a temporal delay is reached. Transitions can be immediate, those that fire in zero time; or temporal, which fire following an exponential distribution whose rate  $\lambda$  is the mean of the distribution. Immediate transitions can have a probability of firing;

they are depicted as black bars while temporal ones are depicted as white bars. This kind of Petri net is useful for the transformation process of UML diagrams at DPIM level; and UML diagrams of Storm applications at DTSM level.

In addition to GSPNs, we include *Stochastic Well-formed colored Nets* (SWN) [21]. A SWN is a Petri net that complements the GSPN extension with data types. Data types are defined as *colors* in the Petri net. The tokens of a certain color only move to compatible places, that is, the colors restrict the places and transitions where a token can be moved. This kind of Petri net is useful for the transformation process of UML diagrams for Hadoop MapReduce applications at DTSM level.

## A.2.2 The Petri Net Markup Language (PNML)

The Petri Net Markup Language (PNML) [37] is an ISO standard for XML-based interchange format for Petri nets. It enables the transformation of the UML diagrams annotated with the DICE profile into an intermediate and standard notation for Petri nets before serializing them into the specific format of a Petri net tool.

The standard contemplates three kinds of Petri Nets: *Place/Transition Nets*, *Symmetric Nets*, and *High-Level Petri Net Graphs* (HLPNGs) as defined in ISO/IEC 15909-1, where *Symmetric Nets* are a restricted version of *High-Level Petri Net Graphs*. In this sense, it is noteworthy to highlight that PNML does not provide native support for any stochastic (e.g., timed) Petri net variant – such as GSPNs and SWNs. Nevertheless, PNML is flexible enough to represent them: the standard provides the extension mechanisms that allow complementing any Petri net element with any non-standard metadata. Thanks to this extensibility, PNML becomes the best candidate for a pivot Petri net format in DICE.

PNML Framework [38] is a free and open-source prototype implementation of the PNML standard, and serves as the reference implementation. It has been designed following the MDE techniques, and as such, a metamodel describing its abstract syntax has been implemented. Figure 29 shows the metaclasses of the PNML Framework metamodel used to define *Place/Transition Nets*. The main Petri net concepts (Places, Arcs, Transitions, etc.) are depicted on the left-hand side of the figure, while the corresponding graphical primitives are depicted on the right-hand side of the figure.

The most relevant elements from the transformation to analysis models point of view are:

**PNML::PetriNetDoc** — A `PetriNetDoc` is a document that meets the requirements of the *PNML Core Model* is called a Petri Net Document. It contains one or more Petri nets (`PetriNet`).

**PNML::PetriNet** — A `PetriNet` consists of one or more `Pages` that in turn consist of several objects. These objects represent the graph structure of the Petri net.

**PNML::Page** — As aforementioned, a `Page` is a container for Petri net objects (`PnObject`). Since a `Page` is an object itself, it may even contain other pages, thus defining a hierarchy of subpages.

**PNML::PnObject** (abstract) — `PnObject` is the base class for any object playing a role in the Petri net structure. Each `PnObject` has a unique identifier which can be used for referring to this object.

**PNML::Node** (abstract) — A `Node` is an entity that can be connected by `Arcs`.

**PNML::PlaceNode** (abstract) — A `PlaceNode` is `Node` representing a place in the Petri net. It can be either a regular `Place` or a `RefPlace`.

**PNML::Place** — A `Place` is `PlaceNode` representing a regular place in the Petri net.

**PNML::RefPlace** — A `RefPlace` is `PlaceNode` that acts as a proxy for a regular `Place`.

**PNML::TransitionNode** (abstract) — A `TransitionNode` is `Node` representing a transition in the Petri net. It can be either a regular `Transition` or a `RefTransition`.

**PNML::Transition** — A `Transition` is `TransitionNode` representing a regular transition in the Petri net.

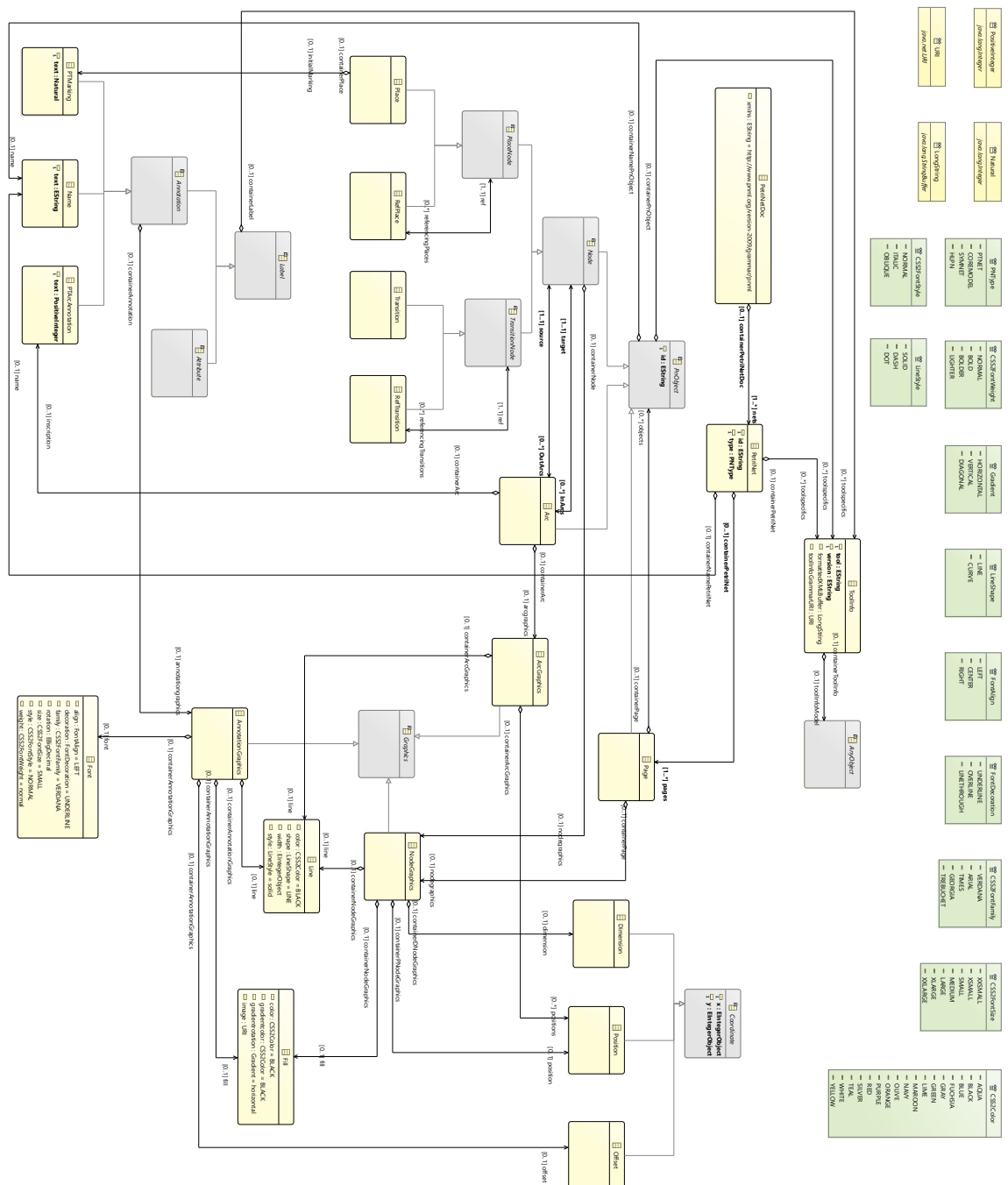


Figure 29: Metaclasses of the Petri Net Markup Language

**PNML::RefTransition** — A RefTransition is TransitionNode that acts as a proxy for a regular Transition.

**PNML::Arc** — An Arc represents a connection from a source Node to a target Node, i.e., from a PlaceNode to a TransitionNode or vice versa.

**PNML::PTMarking** — A PTMarking determines the initial marking of the referenced containerPlace.

**PNML::ToolInfo** — ToolInfo elements store tool specific information. The internal structure of the tool specific information depends on the tool and is not specified by PNML, thus ToolInfo elements can be used to store information not considered by the standard. This is the basic metaelement that we use to represent GSPNs and SWNs using PNML.

## A.3 DTSM Technologies

In this deliverable, we study Hadoop MapReduce and Storm technologies. We have selected them as the initial ones for the transformation process to performance and formal verification models because they are well-established Big Data technologies and they are representatives for processing a set of jobs in batch or streaming mode, which are two of the main processing modes. In particular, we consider UML profile diagrams for Hadoop MapReduce and Storm technologies at DTSM level for the transformation to performance models; and UML profile diagrams for the Storm technology at DTSM level for the transformation to formal verification models. In this section, we present the basic concepts of these technologies that will serve as basis for the definition and comprehension of the transformations in the rest of the document.

### A.3.1 Hadoop MapReduce Basics

The Hadoop MapReduce technology was designed for processing large amounts of datasets in highly parallelizable problems [39]. It extends the map/reduce functions from the functional programming paradigm to cluster environments. Hadoop MapReduce divides the execution of a job into two successive phases, namely *Map* and *Reduce*. The first one consists of the distribution, filtering and sorting of data by a set of small and potentially parallel tasks called mappers. The second phase processes all the partial values and composes the final result. The execution of the reducing phase is carried out by a certain number of tasks called reducers. The tasks are executed in batch mode within a cluster according to the *scheduling policy* selected by the administrator.

More in detail, Hadoop MapReduce is a fault-tolerant technology that automatically relaunches the tasks when they are timed out. Initial data and partial results (intermediate step between the mapping and reducing phases) are stored in a Hadoop Distributed File System (HDFS), which makes the information transparently accessible to all the cluster nodes. A *shuffle* operation efficiently redistributes the output data produced by mappers and send it to the reducers. It takes into account the locality of the data in a working node in order to minimize the transference of data among computers and the network latency.

Hadoop MapReduce has several scheduling policies depending on the organization and assignation of jobs to the resources. By default, all the jobs are piped into a single common FIFO queue for all users (*FIFO Scheduler*) and the cluster runs them in order. In this context, a job (or user) may consume all the available resources without any limitation. The solution proposed in the literature to this starvation problem is two kind of schedulers for multi-user workloads: the *Fair Scheduler* and the *Capacity Scheduler*. They organize jobs into pools, and divide resources fairly between these pools. There is a separate pool for each user (institution) so that each user gets an equal share of the cluster. Within each pool, jobs are scheduled using either fair sharing (Fair Scheduler) or FIFO scheduling (Capacity Scheduler) in order to easily control resource allocation at different granularity levels. Therefore, the computational cores of the cluster are partitioned according to the number of separated pools. The computer cores are assigned to each task depending on the *class of job* (e.g., user identifier) and execution phase. The division of jobs in multiple categories allows the identification and classification of users in the system. In summary, a

Hadoop-based cluster is highly configurable by various parameters ranging from the number of map and reduce tasks to the selection of the scheduling policy.

### A.3.2 Storm Basics

Storm is a distributed real-time computation system for processing large volumes of high-velocity data [40]. In the following we describe those concepts of Storm needed to understand our proposal. A Storm application is usually designed as a directed acyclic graph (DAG) whose *nodes* are the points where the information is generated or processed, and the *edges* define the connections for the transmission of data from one node to another. A Storm application is defined by the topology of the DAG. Two classes of nodes are considered in the topology. On the one hand, *spouts* are sources of information that inject streams of data into the topology. On the other hand, *bolts* elaborate input data and produce results which, in turn, are emitted towards other nodes of the topology. In a generic sense, the topology is a data transformation pipeline. The main difference between Storm and MapReduce technologies is that a Storm application processes the data in real-time while a MapReduce application works with individual batches of tasks. By default, a Storm topology runs indefinitely until killed, while a MapReduce job must eventually end. The notion of *tuples*, *streams* and *messages* are used interchangeably in this part.

A Storm application is configurable by several parameters ranging from the level of parallelism of the nodes (i.e., spouts or bolts) to the multiplicity of the edges (i.e., number of tuples from a certain source that a node requires for producing a message). One of the main characteristics of Storm is the possibility of defining the number of instances or replications (i.e., execution threads) of a node. This value represents the *internal parallelism* of the element and it is constrained by the available number of cores of the computing machine during the deployment. The multiplicity of the edges determines the *message passing*. By default, a message generated by a node of the graph is copied and propagated to every successor in the topology. If the message is received by a bolt with a parallelism greater than 1, it selects to either internally redirect the tuple to any of the multiple instances randomly (*shuffle* policy) or copy the same message to all of them (*all* policy). This policy is determined by the also called *grouping* factor. More grouping options are available, but we consider now only these two policies for simplicity because they are the more representative. The *grouping* policy is assigned to the connection between two nodes.

Figure 30 shows a small Storm topology with a spout (S1) connected to two bolts (B1, B2). S1 has parallelism 1, B1 has parallelism 2 and connection with *shuffle* policy, and B2 has parallelism 3 and connection with *all* policy. The parallelism is represented by N internal buffers for incoming messages where the messages M1, M2 and M3 sent by S1 are placed according to the *grouping* attribute. As spouts and bolts are theoretically independent and asynchronous tasks, the connection between spouts and bolts is buffered. The size of the message queue is not considered because it will not usually influence the model or the performance predictions except for the situations in which the system is saturated. In that

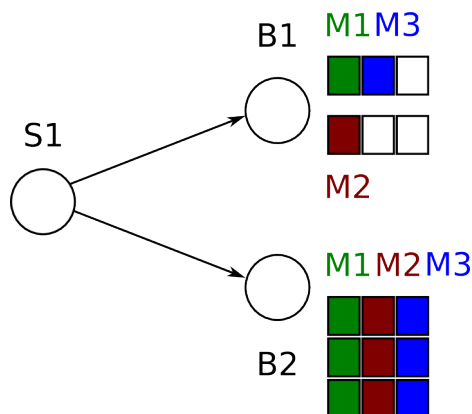


Figure 30: Storm topology with parallelism 2 and shuffle grouping (B1), and parallelism 3 and all grouping (B2)



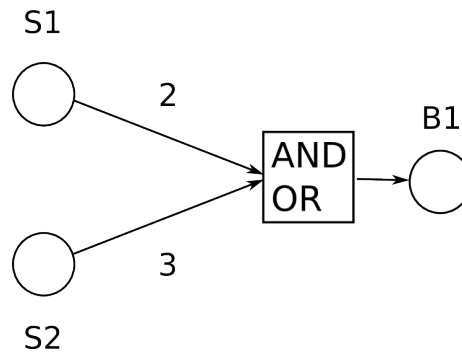


Figure 31: Multiplicity of the connections in Storm

case, it will be enough to limit the representation of the buffer in the performance model. Each buffer has a one-to-one mapping with an execution thread.

A bolt represents a generic processing element that takes inputs and produces outputs. One of the direct consequences of such generality is that each bolt operation may have a different *ratio* of output/input tuples. This asymmetry is captured by the weights in the arcs of the topology. Besides, different *synchronization* policies shall be considered. A bolt receiving messages from two or more sources can select to either 1) progress if at least a tuple from any of the sources is available (*or* policy), or 2) wait for a message from all the sources (*and* policy).

Figure 31 shows a small example of Storm topology with two spouts (S1, S2) connected to a single bolt (B1). The parallelism and grouping parameters are omitted in the image for simplicity. B1 requires 2 tuples from S1 and (or) 3 tuples from S2 for creating an output. The type synchronization is selected by the box.

Finally, the Storm *scheduling algorithm* deploys the components (spouts or bolts) to the computational resources of the cluster statically at the beginning of the execution. The deployment remains unaltered until a failure appears or a rebalance is explicitly requested by the user. By default, the scheduler follows a Round-Robin distribution for mapping the different threads of a component to cluster devices. More complex and user-defined schedulers may take into account the available resources and the software requirements (memory and CPU consumption) for defining an optimal distribution of the tasks. Internally to each computational device, the threads are managed by the OS scheduler. In summary, a Storm topology is highly configurable by various parameters ranging from the internal parallelism of the spouts and bolts to the selection of the message passing policy.





## B MARTE NFP Types

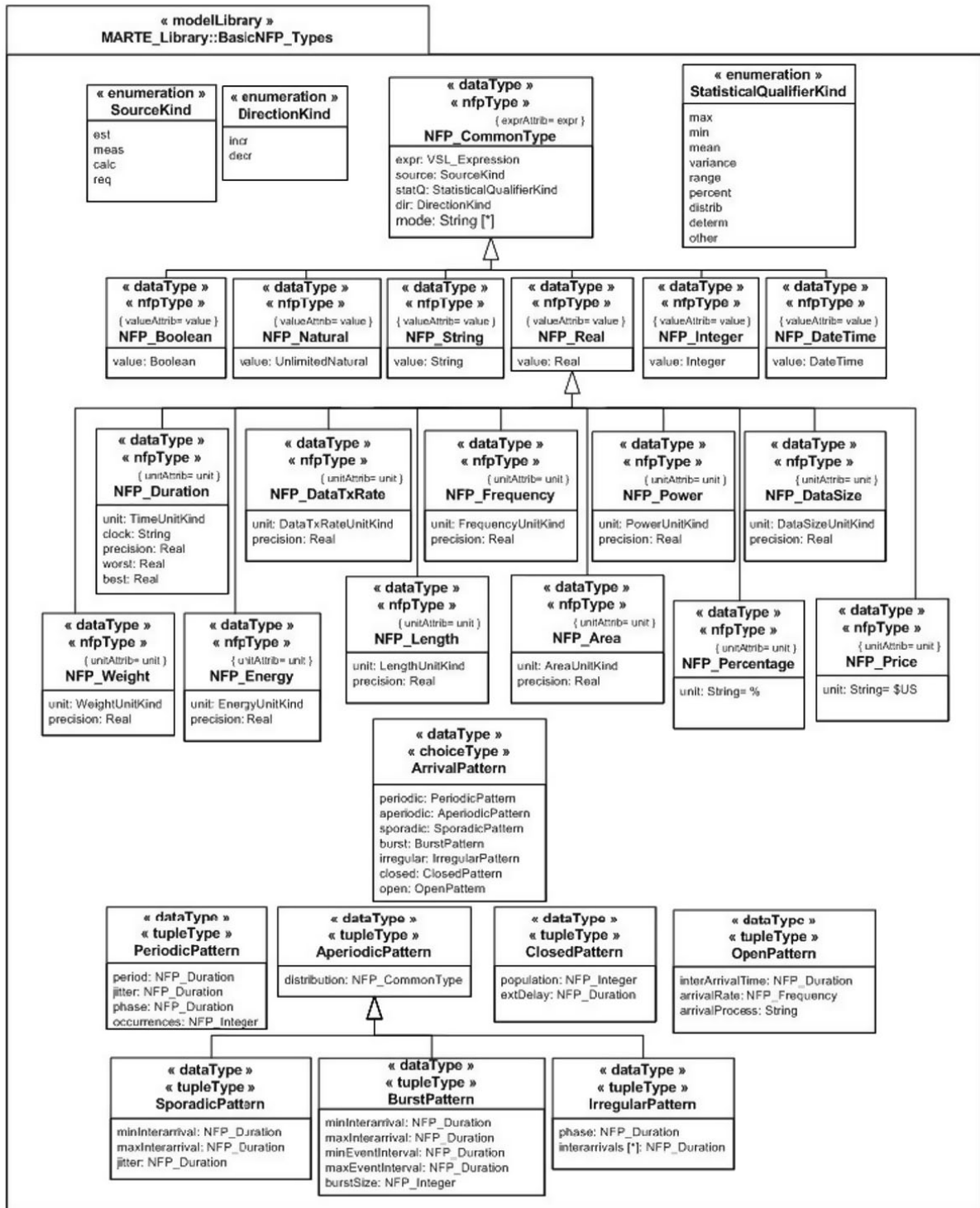


Figure 32: MARTE NFP Types (extracted from [15])

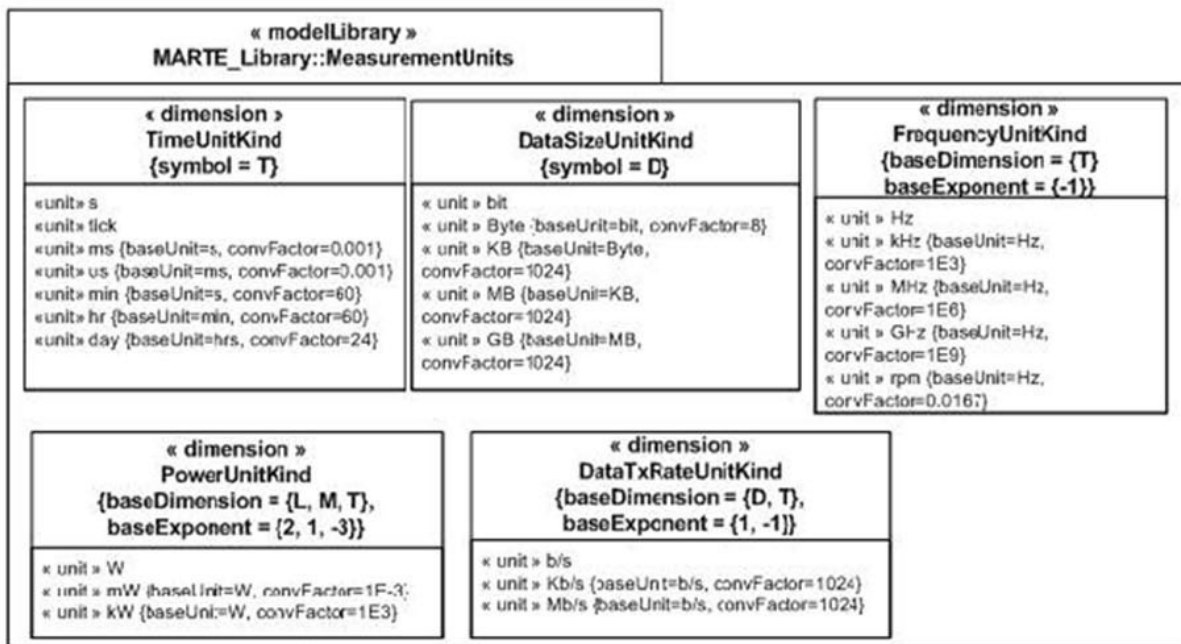


Figure 33: MARTE Measurement Units (extracted from [15])