Ref. Ares(2017)3832092 - 31/07/2017

Developing Data-Intensive Cloud
Applications with Iterative Quality
Enhancements

DICE Verification Tools - Final Version

Deliverable 3.7

Deliverable:	D3.7
Title:	Verification Tools - Final Version
Editor(s):	Francesco Marconi
Contributor(s):	Marcello M. Bersani, Matteo Rossi
Reviewers:	Youssef Ridene, Ioan Dragan
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-Jul-2017
Status:	Final
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

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

Executive summary

This document is the final report on the activities of task T3.3 that focuses on safety verification of data-intensive applications (DIA). All the activities of the task are founded on temporal formalisms that enable the modeling of DIA and the automated analysis of the runtime behavior of the applications by means of temporal logic models. Further details on Task3.3 can be found in D1.1 - State of the art analysis, and D1.2 - Requirements specifications.

Deliverable D3.7 describes both the integration of the **D-VerT** front-end in the DICE IDE and the enhancements applied to the Spark temporal model, already described in deliverable D3.6. The former allow users to define Spark-based DIAs as DICE-profiled UML activity diagrams directly from the DICE IDE, and to automatically run verification tasks on them; the latter allow **D-VerT** to perform a faster verification than the one carried out using the previous modeling of D3.6.

Working on efficiency aspects turned out to be fundamental to increase the usability of the verification tool that, as usual in this context, might suffer from state explosion that worsen the performance. Improving the modeling makes the automated analysis of the DIA models more viable in terms of time cost and, therefore, more practicable for users. The document briefly reports on some aspects of the temporal logic model of Spark applications and focuses on the improvements to the model that are implemented in **D-VerT**. The main verification approach adopted for verification in DICE is based on satisfiability checking of temporal formulae that relies on an engine solving the satisfiability problem called **Zot**[1]. Being the procedure based on SMT-solvers, the two fundamental modifications investigated aim at reducing the size of the formula and the size of its solutions.

The document also describes the UML representation that the user can use to specify the Spark application. Activity diagrams are the adopted solution that allow the modeling of a Spark application by means of a high level graphical language. Basic functional blocks corresponding to Spark transformations and actions are available to the user that can design the application with a compositional approach.

[1]`https://github.com/fm-polimi/zot`

Glossary

CLTLoc	Constraint Linear Temporal Logic over clocks
DIA	Data-Intensive Application
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DPIM	DICE Platform Independent Model
DTSM	DICE Platform and Technology Specific Model
FOL	First-order Logic
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
M2M	Model to Model transformation
QA	Quality Assurance
TL	Temporal Logic
UML	Unified Modelling Language

Contents

Executive summary . . . **3**

Glossary . . . **4**

Table of Contents . . . **5**

List of Figures . . . **6**

List of Tables . . . **6**

List of Listings . . . **6**

1 Introduction . . . **7**
- 1.1 Objectives . . . 8
- 1.2 Motivation . . . 8
- 1.3 Structure of the deliverable . . . 9

2 Requirements and usage scenarios . . . **10**
- 2.1 Tools and actors . . . 10
- 2.2 Use cases and requirements . . . 10

3 Modeling Spark Applications for Verification . . . **13**
- 3.1 Spark basics . . . 13
- 3.2 Modeling assumptions and Job model . . . 13
  - 3.2.1 Assumptions . . . 14
- 3.3 DICE Profile for Spark design . . . 14
- 3.4 UML design for verification . . . 14

4 Integration in the DICE IDE . . . **17**
- 4.1 Transformation from UML diagram to formal model . . . 17
- 4.2 Launch configuration type . . . 19

5 Improvements and optimizations on formal model and tool . . . **20**
- 5.1 Encoding . . . 20
  - 5.1.1 CLTLoc Spark job modeling . . . 20
- 5.2 Model size reduction by means of labeling . . . 22

6 Use case . . . **24**

7 Remark on the use of Containerizing technologies . . . **26**

8 Conclusions . . . **26**
- 8.1 Upcoming work . . . 26

References . . . **27**

List of Figures

1 Stereotypes from the DICE::DTSM::Spark profile that are needed to model Spark DIAs for verification. 15
2 UML representation of word-count application 16
3 Two-steps transformation flow of **D-VerT**: the DIA, designed as a DICE-profiled UML diagram (1), is first translated to a JSON object (2) and then to the corresponding Temporal Logic model (3). 17
4 UML Class diagram showing the hierarchy of nodes, each of them encapsulating UML activity nodes. 18
5 Class diagram showing the classes used to implement the transformation. 18
6 Launch configuration for Spark verification in DICE 19
7 Linear DAG 23
8 DAG generated for the k-Means application. Labels are represented between angle brackets. 23
9 UML Activity diagram of the use case application. 25
10 Labeled execution DAG generated from the use case application. 25

List of Tables

1 Requirement coverage at month M30. R3.15 is coverd or 90% as the privacy concern requirements have been moved to Trace-checking analysis in Work Package 4 (details about privacy are supplied by deliverable D4.4 and D2.4). 27

List of Listings

1 Python implementation of the word-count application. 15
2 Python code representing the example Spark application. 24

1 Introduction

The *DICE Verification Tool* (**D-VerT**) is designed for the analysis of safety aspects of data-intensive application. It allows designers to evaluate the design against safety properties expressed with a temporal language (such as reachability of undesired configurations of the system, meeting of deadlines, and so on) based on the very well-known Linear Temporal Logic (LTL). The outcome of the verification task performed through the use of **D-VerT** is employed by the designer of an application for refining its model at design time, in case anomalies are detected.

Verification is commonly considered an hard task. The theoretical complexity of verification algorithms is, in general, not suitable for the analysis even for small systems. However, in some cases where the complexity of verification is considered tractable in practice, the state space explosion is still the most significant barrier hampering the feasibility of the analysis. The state space explosion often leads the execution time of verification procedures to a dramatic growth. Many and various are the techniques and approaches to limit this disadvantage. Verification in DICE is realized through a bounded approach stemming from the well-know bounded model-checking, a verification approach that turned out to be effective for limiting the cost of the analysis in many practical scenarios. This verification approach was originally devised to identify violations of a property of a system that can be described by means of a representation of limited size. The verification procedure does not look for a possible violation candidate in the set of all the violating executions of the system but it does operate the search in a smaller set, i.e., the set of all the violating executions that can be represented by means of a representation smaller than a given size.

In many situations however, the time expended to solve the instances of a verification problem that can be found in realistic scenario is still high, despite the adoption of a bounded verification approach. In Deliverable D3.6, we presented a temporal logic model of the execution of a Spark job that can be suitable for verifying the overall time span of the computation. The model considers that jobs run on a cluster with a finite number of computational resources and that all the computations terminate (i.e., the number of loop iterations is limited). The temporal logic model of Spark was used in a first experimental campaign to test its usability in terms of time to perform the analysis. The experimental results showed however that the time to verify an excerpt of a possible execution of a Spark application, i.e., determining the existence of executions that violate specific temporal properties, is high and not suitable for practical use. Deliverable D3.7 provides a first analysis of this issue and it introduces two techniques that can be adopted to lower the solving time of the verification. Both of them rely on the same principle according to which the resolution time of a problem can be reduced by reducing the size of the representation of both the instance of the problem and the solution. To this end, the two means that were implemented are the following.

- To reduce the size of model, being it based on logic, we devise a way to diminish the number of atomic propositions that appear in the formulae. This avoids the use of a large set of atoms, by fostering the reuse of the same atom to model more than one system event or state.
- A solution of a temporal formula is either unsat or a finite sequence of system events that violates a property. In general, each position of the sequence is associated with an event of the system. To reduce the size of the solution some modifications still affected the logical model and allowed the representation of more than one event of the system for each single position of the execution trace.

The second contribution of Deliverable D3.7 the translation of UML activity diagrams, that we use to represent Spark applications, into a temporal logic formula. Spark applications are implemented by means of basic functional blocks, called transformations and actions, that perform specific functionality on their input data (e.g., filtering, functional mapping, etc.). An activity diagram can be adopted by the DICE designer to structure the application and to define the flow of computation by means of a sequence of Spark transformations that can be executed in parallel and that are concluded by an action. This sequence of operations is then automatically transformed into the actual execution DAG, which is composed of *stages* (sets of pipelined operations that can be executed in parallel). As described in [1], the temporal logic model derives from the execution DAG. Therefore, the automatic translation of the

UML diagram allows the analysis of the user design by hiding the technical details of the underlying verification engine.

D-VerT is published as an open source tool in the DICE-Verification repository of the project Github[2].

1.1 Objectives

The main achievement of Work Package 3 (WP3) is the development of a quality analysis tool-chain that supports the following analysis:

(i) simulation-based assessment for reliability and efficiency,

(ii) formal verification of safety properties, and

(iii) numerical optimization techniques for the search of optimal architecture designs.

Task T3.3 is related to point (ii) and concerns the following issues.

- Task T3.3 works towards a verification framework that enables automatic evaluation of safety properties of DIAs, limited to Storm topologies and Spark jobs.
- Verification is carried out through satisfiability checking of Constraint LTL over-clocks (CLTLoc) formulae, that represent (an abstraction of) the DIA behavior over time. The safety analysis is carried out at the DTSM level of the DICE design workflow.
- The outcome of the verification task allows the designers to analyze whether the system properties are satisfied, and not where the problem occurred and how to fix it. In the case of violations, the output trace gives a hint to the designer on what should be fixed.

The work undertaken in the last period of activity in Task T3.3 has been focused on the following activities.

1. *Modeling Spark application in UML.* The activity investigated the use of UML diagrams as specification languages to be adopted at design time for the specification of a Spark application. The result of this work is described in Section 3.
2. *Integration of **D-VerT** in the DICE framework.* The activity has been carried on in order to add the Spark technology in **D-VerT** and allow users to run verification for Spark applications.
3. *Definition of strategies for improving performance of verification.* The activity provided **D-VerT** with a refined encoding of Spark jobs that results in a faster verification tasks. The result of this work is described in Section 5.

1.2 Motivation

The analysis of correctness is fundamental to produce systems that behave correctly at runtime. Verification in DICE aims to define the meaning of correctness for DIAs and provides implementation of tools supporting formal analysis of DIAs. Task T3.3 is motivated by this need and promotes safety verification of DIAs through the use of **D-VerT**.

Verification in DICE relies on a fully automatic procedure that is based on dense-time temporal logic and it is realized in accordance with the bounded model-checking approach. A common limitation of formal verification in general is the high cost in term of time required to perform verification. Therefore, to exploit formal analysis in DICE, the research of efficient procedure has been supported at all levels, from the modeling to the underlying solvers used to carry out verification. In particular, this document

[2]`https://github.com/dice-project/DICE-Verification`

focuses on the definition of some refinements of the Spark temporal model that was presented the first time in Deliverable D3.6. The effectiveness of the new model has been observed through experimental results showing the benefits in terms of resolution time.

1.3 Structure of the deliverable

Section 2 outlines the main requirements of the verification solution achieved by **D-VerT**. Section 3 summarizes the main concepts about the behavior of Spark at runtime, shows the assumptions that allowed the definition of the temporal logic model of Spark jobs and recalls the fundamental UML objects in the DICE profile that enables the verification of Spark jobs, bridging WP2's models with verification goals. Section 4 outlines the main steps enforced to integrate the new features in **D-VerT**. Section 5 elaborates on the improvements that are implemented in the latest version of the temporal logic model of Spark jobs. Section 6 provides a use case for validating the new features of **D-VerT**. Section 7 comments on the use of containerizing technologies with respect to the verification approach adopted so far. Section 8 draws the conclusions about the work realized by WP3 - Verification unit.

2 Requirements and usage scenarios

Deliverable D1.2 [2, 3] presents the requirements analysis for the DICE project. The outcome of the analysis is a consolidated list of requirements and the list of use cases that define the project's goals.

This section summarizes, for Task T3.3, the requirements and the use case scenarios and explains how they have been fulfilled in the current **D-VerT**.

2.1 Tools and actors

As specified in D1.2, the data-aware quality analysis aims at assessing quality requirements for DIAs and at offering an optimized deployment configuration for the application. The assessment elaborates DIA UML diagrams, which include the definition of the application functionalities and suitable annotations, including those for verification, and employs the following tools:

- Transformation Tools
- Simulation Tools
- Verification Tools — **D-VerT**, which takes the UML models produced by the application designers, and verifies the safety and privacy requirements of the DIA.
- Optimization Tools

In the rest of this document, we focus on the tools related to Task T3.3, i.e., **D-VerT**. According to deliverable D1.2 the relevant stakeholders are the following:

- **QA_ENGINEER** — The application quality engineer uses **D-VerT** through the DICE IDE.
- **Verification Tool** (**D-VerT**) — The tool invokes suitable transformations to produce, from the high-level UML description of the DIA, the formal model to be evaluated. It is built on top of two distinct engines that are capable of performing verification activities for temporal logic-based models and FOL-based models, respectively. Such tools are invoked according to the QA_ENGINEER needs. We later refer to them as TL-solver and FOL-solver, respectively.

2.2 Use cases and requirements

The requirements elicitation of D1.2 considers a single use case that concerns **D-VerT**, namely UC3.2. This use case can be summarized as follows [2, p.104]:

ID:	UC3.2
Title:	Verification of safety and privacy properties from a DICE UML model
Priority:	REQUIRED
Actors:	QA_ENGINEER, IDE, TRANSFORMATION_TOOLS, VERIFICATION_TOOLS
Pre-conditions:	There exists a UML model built using the DICE profile. A property to be checked has been defined through the DICE profile, or at least through the DICE IDE, by instantiating some pattern.
Post-conditions:	The QA_ENGINEER gets information about whether the property holds for the modelled system or not

The requirements listed in [2] are the following:

ID:	R3.1
Title:	M2M Transformation
Priority of accomplishment:	Must have
Description:	The TRANSFORMATION_TOOLS MUST perform a model-to-model transformation, [...] from DPIM or DTSM DICE annotated UML model to formal model.

ID:	R3.2
Title:	Taking into account relevant annotations
Priority of accomplishment:	Must have
Description:	The TRANSFORMATION_TOOLS MUST take into account the relevant annotations [...] and transform them into the corresponding artifact [...]

ID:	R3.3
Title:	Transformation rules
Priority of accomplishment:	Could have
Description:	The TRANSFORMATION_TOOLS MAY be able to extract, interpret and apply the transformation rules from an external source.

ID:	R3.7
Title:	Generation of traces from the system model
Priority of accomplishment:	Must have
Description:	The VERIFICATION_TOOLS MUST be able [...] to show possible execution traces of the system [...]

ID:	R3.10
Title:	SLA specification and compliance
Priority of accomplishment:	Must have
Description:	VERIFICATION_TOOLS [...] MUST permit users to check their outputs against SLA's [...]

ID:	R3.12
Title:	Modelling abstraction level
Priority of accomplishment:	Must have
Description:	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

ID:	R3.15
Title:	Verification of temporal safety/privacy properties
Priority of accomplishment:	Must have
Description:	[...] the VERIFICATION_TOOLS MUST be able to answer [...] whether the property holds for the modeled system or not.

ID:	R3IDE.2
Title:	Timeout specification
Priority of accomplishment:	Should have
Description:	The IDE SHOULD allow [..] to set a timeout and a maximum amount of memory [...] when running [...] the VERIFICATION_TOOLS. [...]

ID:	R3IDE.4
Title:	Loading the annotated UML model
Priority of accomplishment:	Must have
Description:	The DICE IDE MUST include a command to launch the [...] VERIFICATION_TOOLS [...]

ID:	R3IDE.4.1
Title:	Usability of the IDE-VERIFICATION_TOOLS interaction
Priority of accomplishment:	Should have
Description:	The QA_ENGINEER SHOULD not perceive a difference between the IDE and the VERIFICATION_TOOL [...]

ID:	R3IDE.4.2
Title:	Loading of the property to be verified
Priority of accomplishment:	Must have
Description:	The VERIFICATION_TOOLS MUST be able to handle [...] properties [...] defined through the IDE and the DICE profile

ID:	R3IDE.5
Title:	Graphical output
Priority of accomplishment:	Should have
Description:	[...] the IDE SHOULD be able to take the output generated by the VERIFICATION_TOOLS [...]

ID:	R3IDE.5.1
Title:	Graphical output of erroneous behaviors
Priority of accomplishment:	Could have
Description:	[...] the VERIFICATION_TOOLS COULD provide [...] an indication of where in the trace lies the problem

3 Modeling Spark Applications for Verification

This section elaborates on how Spark applications can be modeled by means of DICE-profiled UML activity diagrams.

We already mentioned in [1] all the assumptions that have been made with respect to the modeling of Spark applications. A brief excerpt from [1] about the main features of the framework modeled for the verification and the assumptions underlying the modeling is provided afterwards.

3.1 Spark basics

Spark is a framework that allows developers to implement DIAs that process data streams or batches and run on clusters of independent computational resources. The computational model of Spark is specifically designed to guarantee data parallelism and fault-tolerant executions. Data are uniformly partitioned across nodes and multiple partitions can be concurrently processed by applying the same operations in parallel. Spark allows the development based on two types of operations:

- *Transformations* are operations (such as map, filter, join, union, and so on) that are performed on data and which yield new data.
- *Actions* are operations (such as reduce, count, first, and so on) that return a value obtained by executing a computation on data.

Transformations in Spark are "lazy" as they do not compute their results immediately upon a function call. Spark arranges the transformations to maximize the number of such operations executed in parallel by scheduling them in a proper way. It keeps track of the dataset that the transformation operates and computes the transformations only when an action is called.

As the code is submitted to it, Spark creates the so-called *operator DAG* (or *RDD DAG*), whose nodes are the operations performed over data. This graph keeps track also of all the RDDs that are created as consequence of the operations.

The *operator graph* is then submitted to the Spark *DAG Scheduler*. The DAG scheduler pipelines operators together when possible, and creates what we will refer to as the *execution DAG*, whose nodes are *stages*.

A *stage* is a sequence of transformations that are performed in parallel over many partitions of the data and that are generally concluded by a shuffle operation. Each stage is a computational entity that produces a result as soon as all its constituting operations are completed. Each stage consists of many tasks that carry out the transformations of the stage; a *task* is a unit of computation that is executed on a single partition of data. The computation realized by a DAG is called *Spark job*, i.e., an application which reads some input, performs some computation and returns some output data. A DAG defines the functionality of a Spark job by means of an operational workflow that specifies the dependencies among the stages manipulating RDDs. The dependency between two stages is a precedence relation. Hence, a stage can be executed only if all its predecessors have finished their computation.

3.2 Modeling assumptions and Job model

The verification of Spark jobs is carried out on the DAG underlying the application and it is based on an abstraction of the temporal behavior of the tasks implementing the stages. The logical model characterizes each task with

- a latency that is an estimation of the duration of the task per data partition
- the number of CPU cores executing the task.

The data partitions are obtained by the Spark engine when the Spark job is launched and the partition size is a parameter that can be set before launching the application. For this reason, the DAG of the

Spark job, the number of tasks per stage and number of cores are required parameters to instantiate the analysis.

Verification is performed by means of a logical model written in CLTLoc. The CLTLoc model represents the execution of a DAG of stages over time. In particular, it describes their activation, execution and termination with respect to the precedence relation among the stages, that is entailed by the DAG of the application. The prerequisite for the analysis is the availability of the task latency required to perform the Spark operators occurring in the DAG, as each task carries out the operation of a Storm operator. Verification concerns non-functional properties of the DIAs and the task functionalities are abstracted away with their timing requirements.

3.2.1 Assumptions

Spark environment

- The runtime environment that Spark instruments to run a job is not considered in the modeling.
- The latency generated by the execution of services managing the jobs is considered negligible with respect to the total time for executing the application.

Cluster environment

- The workload of the cluster executing the application is not subject to oscillations that might alter the execution of the running jobs.
- The cluster performance is stable and does not vary over time.

Spark job

- The number of CPU cores that are available for computing the Spark job is known before starting the execution of the job and does not vary over the computation.
- All the stages include a finite number of identical tasks, i.e., the temporal abstraction that models their functionalities is the same; therefore, all the tasks constituting a stage have durations that can vary non-deterministically by at most a fraction of the nominal stage duration.

3.3 DICE Profile for Spark design

The modeling of Spark applications for verification benefits from the features provided by the DICE Profiles [4]. The profile includes all the stereotypes to define the main functional and non-functional details of the elements constituting DIAs implemented in Spark. The set of stereotypes needed to enable verification is shown in Fig. 1. Specifically, $<< SparkScenario >>$ allows the designer to define some general information about the application and its context, such as the default parallelism adopted by Spark, the number of cores and the quantity of memory of the underlying cluster that are assigned to the application. $<< SparkMap >>$ and $<< SparkReduce >>$ provide the features to define, respectively, transformations and actions. The attribute `MapType` (`ReduceType`, respectively) is used to specify the kind of operation that is performed. Other attributes, such as `duration` and `numTasks` define the reference duration of each operation with respect ot a single unit of data (partition) and the level of partitioning (if specified) that will be applied on the input data to perform such operation.

3.4 UML design for verification

Spark applications can be designed from the DICE IDE as UML Activity Diagrams. The main building blocks provided by the Papyrus editor are the following:

- **Activity node**: acts as a "container" of the Spark application, as all the nodes of the operations DAG need to be included in it. By applying on it the $<< SparkScenario >>$ stereotype, the user can define the context of the application.

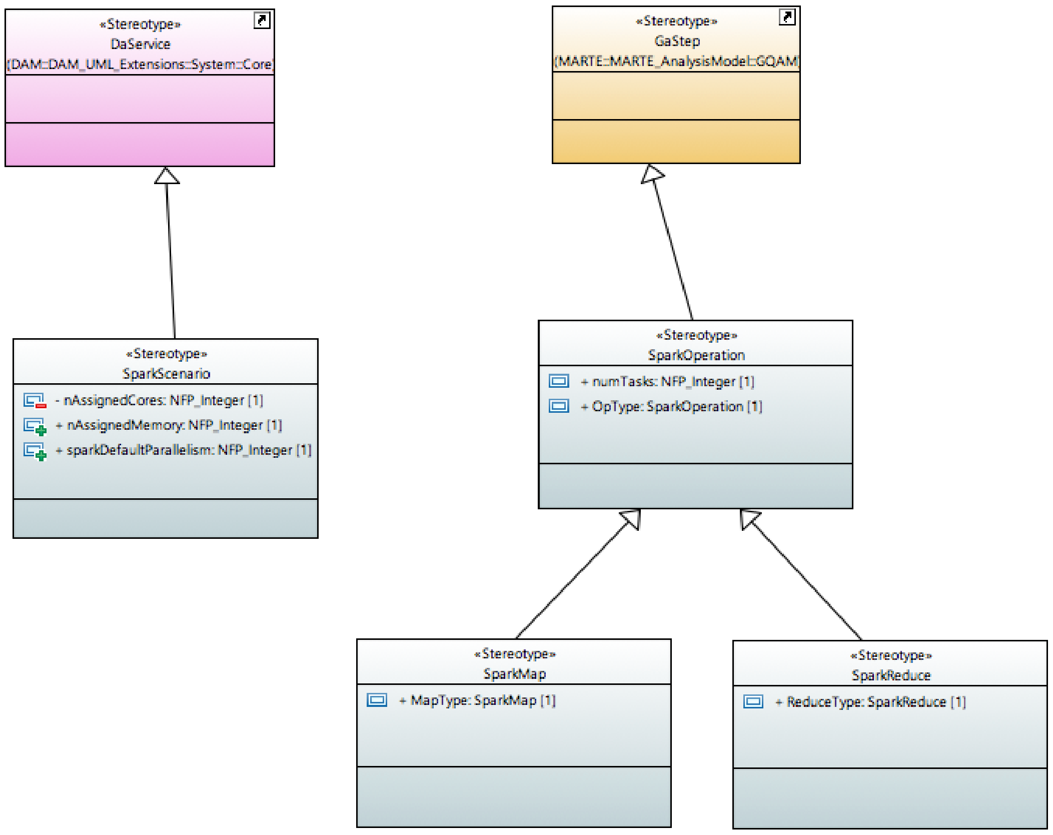

«Stereotype»
DaService
(DAM::DAM_UML_Extensions::System::Core
«Stereotype»
GaStep
(MARTE::MARTE_AnalysisModel::GQAM)
«Stereotype»
SparkScenario
- nAssignedCores: NFP_Integer [1]
+ nAssignedMemory: NFP_Integer [1]
+ sparkDefaultParallelism: NFP_Integer [1]
«Stereotype»
SparkOperation
+ numTasks: NFP_Integer [1]
+ OpType: SparkOperation [1]
«Stereotype»
SparkMap
+ MapType: SparkMap [1]
«Stereotype»
SparkReduce
+ ReduceType: SparkReduce [1]

Figure 1: Stereotypes from the DICE::DTSM::Spark profile that are needed to model Spark DIAs for verification.

- **Opaque Action nodes**: are used to specify generic Spark operations. The user, by applying the specific stereotype (which can be either $<< SparkMap >>$ or $<< SparkReduce >>$) can characterize further the operation as a transformation or action.
- **Control Flow edges**: are used to connect the various operations (Opaque Action nodes) to compose the operator DAG.
- **Initial Nodes**: express the starting points of the computation and the initialization point for RDDs.

Consistently with how applications are defined to run the simulation tools [5], the semantics that is considered for the Activity diagrams is slightly different from the standard UML semantics, in which there can be only one initial node and one end node for each activity: the user can define more than one initial node, as each one of them corresponds to the initialization of an RDD that is used as a source of computation.

In order to clarify the design process, we are showing the representation of a very simple application– the classic "word-count" example.
The word-count application consists of a series of steps that are performed in order to obtain, given an input text, the number of occurrences of each different word on that text. Listing 1 shows the few lines of Python code that are needed to define the application.

Listing 1: Python implementation of the word-count application.

```
text = sc.textFile("s3://...")
words = text.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y).collect()
```

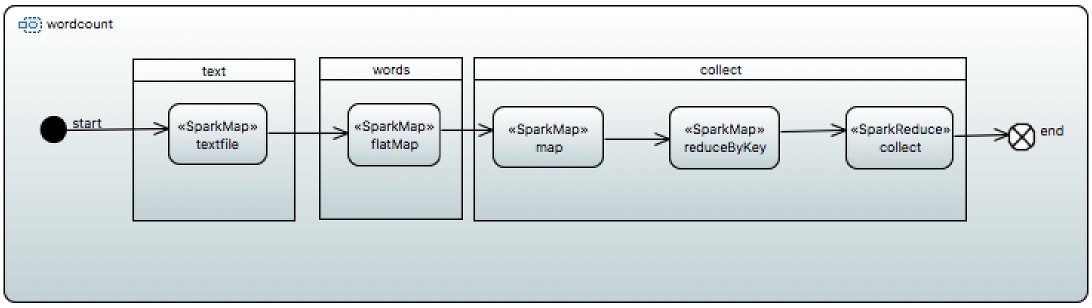


Figure 2: UML representation of word-count application

As can be seen from the code, the first step consists in reading from file with the `textfile()` transformation, which produces the `text` RDD (containing the whole text to analyze). Then a `flatmap()` is applied on `text` to split the text in words. Subsequently, each word `x` is mapped to a tuple `(x, 1)` by means of a `map()` transformation. Finally the `reduceByKey()` transformation computes the count of the occurrences for each word. The `collect` action triggers the execution of the previous steps as a Spark job and allows for the collection of the final outcome on the driver machine.

The corresponding UML description of the application is depicted in Fig. 9. The activity node called *wordcount* contains the operator DAG and is tagged with the $<< SparkScenario >>$ stereotype. The various computation steps (*textFile*, *flatMap*, *map*, *reduceByKey*, *collect*) are inserted as Opaque Action nodes and are tagged with the appropriate stereotype: *collect*, being an action, is tagged with the $<< SparkReduce >>$ stereotype, while the others are all tagged as $<< SparkMap >>$. Each operation is then configured properly by setting the specific *MapType* (or *ReduceType*) as well as the other required parameters, such as *duration* and *numTask*. The activity partitions shown in Fig. 2 highlight how the variables defined in the code are mapped over the different nodes of the Activity diagram. Their usage is exclusively for illustrative purpose and they do not provide any semantic value that will be considered for the transformations from UML to temporal logic.

4 Integration in the DICE IDE

This section reports the enhancements that have been done regarding the integration of **D-VerT** in the DICE IDE.

The front-end of **D-VerT**, consisting of a set of Eclipse plugins, was already integrated in the DICE IDE, as described in [6]. However, the previous version of the tool only supported the verification of Storm DIAs from the front-end. Therefore, most of the effort was dedicated to enabling Spark verification starting from UML models.
The support for Spark verification was made possible by:

1. the definition of a UML semantics for designing Spark application, provided by the updated DICE Profile and by the modeling criteria discussed in Sect. 3;

2. the creation a new, dedicated, launch configuration dialog (*DICE Verification - Spark*);

3. the implementation of the transformation from DICE-profiled UML diagram to the formal model;

4.1 Transformation from UML diagram to formal model

The transformation from the DICE-profiled Spark UML diagram to the formal model has been implemented by extending the approach already adopted in **D-VerT** to support Apache Storm technology.

As reported in [7], **D-VerT** defines a two-steps transformation process (Fig. 3): the first step, carried out on the front-end by the **Uml2Json** component, translates the UML diagram to a specific JSON description of the DIA; the second step, implemented in the back-end through the **Json2MC** component, which generates the specific instance of the formal model by based on the content of the JSON object received as input.

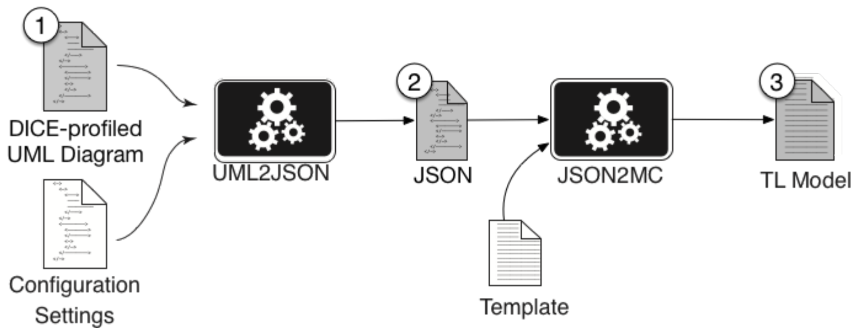
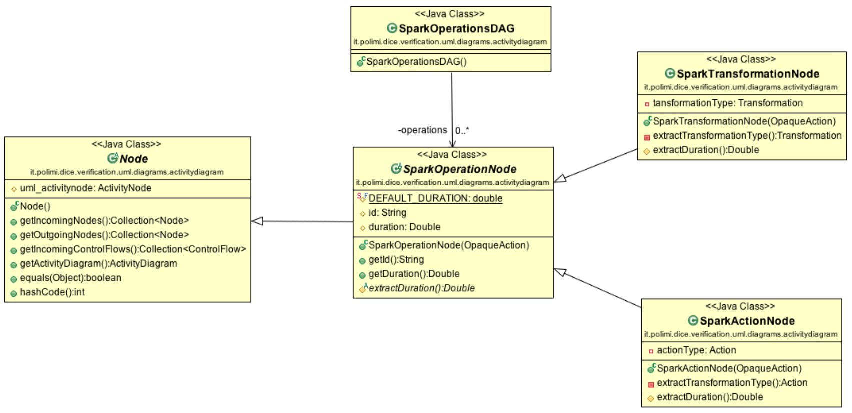


Figure 3: Two-steps transformation flow of **D-VerT**: the DIA, designed as a DICE-profiled UML diagram (1), is first translated to a JSON object (2) and then to the corresponding Temporal Logic model (3).

The **Uml2Json** component, part of the **D-VerT** Eclipse plugins, is entirely developed in Java and relies on the Eclipse UML2[3] Java Library, an EMF-based implementation of the UML 2.x OMG metamodel for the Eclipse platform. **Uml2Json** traverses the XMI file containing the diagram and extract all the needed information, instantiating the data structures to perform the transformation, shown in Fig. 4 and Fig. 5. As presented in Fig. 4, abstract class *Node* encapsulates a generic node of an activity diagram and provides some basic operations. The *SparkOperationNode* abstract class extends it, defining the concept of a generic Spark operation and providing the methods which are common to all the operations. *SparkTransformationNode* and *SparkActionNode* are the concrete classes representing the two kinds of Spark operations. They provide all the functionalities to extract the specific values defined in the corresponding stereotypes. While traversing the UML diagram, **Uml2Json** decorates each occurrence of an Opaque Action node with the appropriate concrete *SparkOperation* node, depending on the applied stereotype. Each node keeps track of its connected nodes and is added to a *SparkOperationsDAG* instance.

[3]`http://www.eclipse.org/modeling/mdt/?project=uml2`



<<Java Class>>
SparkOperationsDAG
it.polimi.dice.verification.uml.diagrams.activitydiagram
SparkOperationsDAG()
-operations 0..*
<<Java Class>>
SparkTransformationNode
it.polimi.dice.verification.uml.diagrams.activitydiagram
tansformationType: Transformation
SparkTransformationNode(OpaqueAction)
extractTransformationType():Transformation
extractDuration():Double
<<Java Class>>
Node
it.polimi.dice.verification.uml.diagrams.activitydiagram
uml_activitynode: ActivityNode
Node()
getIncomingNodes():Collection<Node>
getOutgoingNodes():Collection<Node>
getIncomingControlFlows():Collection<ControlFlow>
getActivityDiagram():ActivityDiagram
equals(Object):boolean
hashCode():int
<<Java Class>>
SparkOperationNode
it.polimi.dice.verification.uml.diagrams.activitydiagram
DEFAULT_DURATION: double
id: String
duration: Double
SparkOperationNode(OpaqueAction)
getId():String
getDuration():Double
extractDuration():Double
<<Java Class>>
SparkActionNode
it.polimi.dice.verification.uml.diagrams.activitydiagram
actionType: Action
SparkActionNode(OpaqueAction)
extractTransformationType():Action
extractDuration():Double

Figure 4: UML Class diagram showing the hierarchy of nodes, each of them encapsulating UML activity nodes.

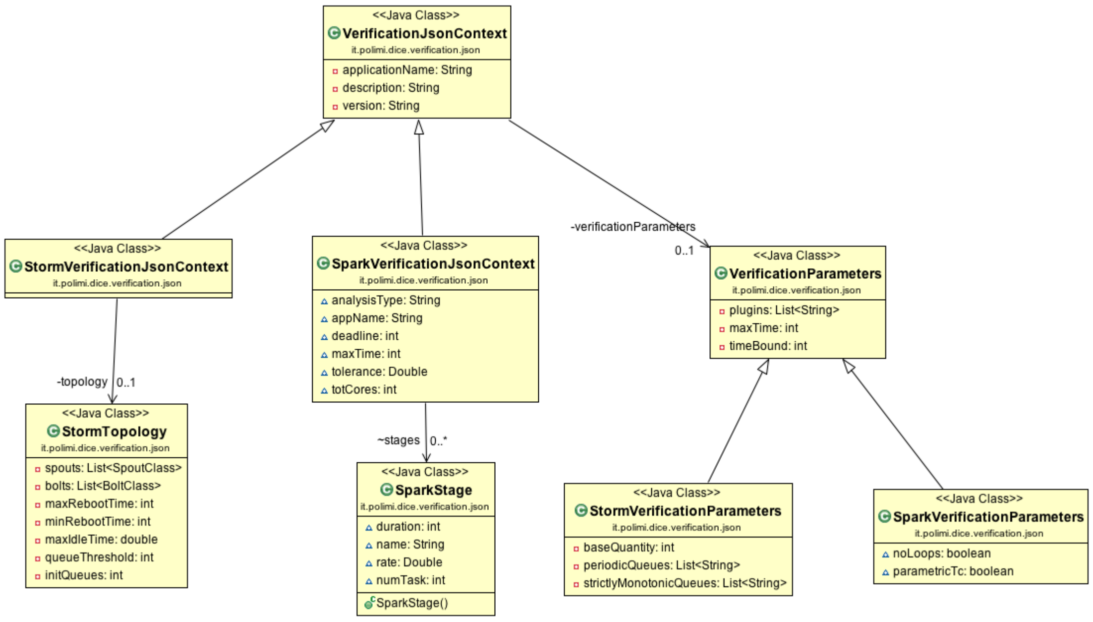
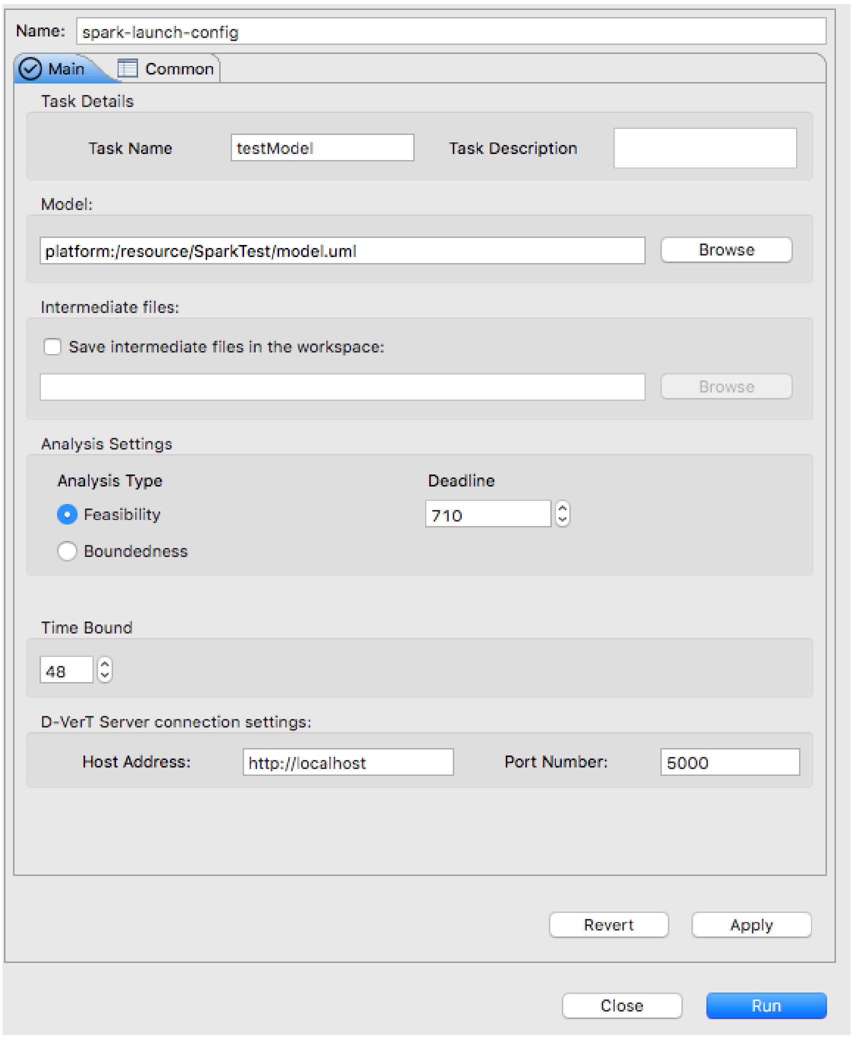


Figure 5: Class diagram showing the classes used to implement the transformation.

Differently from the transformation from Storm UML class diagrams to JSON, in which the translation was merely structural, Spark operator DAGs need to be transformed by producing the corresponding Spark execution DAGs, whose nodes consist in *stages*. For this reason **Uml2Json** needs to mimic the same policies that are used by Spark to create the execution DAG. Once the *SparkOperationsDAG* instance is instantiated, it is analyzed in order to create the set of stages that will form the execution DAG. The creation of a new stage, consisting in the instantiation of the *SparkStage* class of Fig. 5, is done whenever a *shuffle* transformation is encountered. A stage is typically composed of many operations which are pipelined and executed in sequence by each thread, therefore the duration parameter of the stage (i. e. the reference duration of each task in the stage) consists in the sum of the durations of all the operations in the stage. The list of stages is included in an instance of the *SparkVerificationJsonContext* class (Fig. 5) together with the configuration provided through the run configuration. At this point, all the information needed to proceed with the transformation is put together and serialized as a JSON Object by means of the *gson*[4] Java library. The JSON Object is then used to build a POST request to the **D-VerT** server which will trigger the second step of the transformation and consequently the launch of the verification task. More details about the generation of the formal model starting from the JSON object can be found in [1].

[4] `https://github.com/google/gson`



Name: spark-launch-config

Task Details

Task Name: testModel
Task Description:

Model: platform:/resource/SparkTest/model.uml

Intermediate files:

[ ] Save intermediate files in the workspace:

Analysis Settings

Analysis Type: [✓ Feasibility] Boundedness

Deadline: 710

Time Bound: 48

D-VerT Server connection settings:

Host Address: http://localhost
Port Number: 5000

Figure 6: Launch configuration for Spark verification in DICE

4.2 Launch configuration type

As for Storm verification, the transformation process is triggered from the IDE by means of a specific launch configuration type, which provides all the necessary tool settings to perform formal verification.

The **D-VerT** Spark launch configuration, as shown in Fig. 6, allows the user to select the input model, select the analysis type to perform, configure the deadline against which perform the analysis and specify the time bound over which the verification has to be carried out. Additionally, it is possible to add textual informations about the verification task instance, save the intermediate files in a local folder and configure the network address of the **D-VerT** server.

The launch configuration was developed in Java exploiting the Eclipse Launching framework.

5 Improvements and optimizations on formal model and tool

This section introduces some techniques to lower the cost of performing the verification tasks on Spark models. More precisely, it introduces the following strategies for improvements:

- Reduction of the size of the the model by using a "labeling" mechanism on the DAG.
- Reduction of the size of the possible output traces to be returned by aggregating sequences of repetitive events.

5.1 Encoding

As already emphasized in the introduction, the research for approaches enhancing the performance of automatic procedures, suitable for solving complex problems, becomes fundamental to improve the usability of the implementing tools in practical contexts, where the complexity of the scenarios could make the use of such tools too onerous. In DICE verification, where the analysis relies on of logic-based techniques, the reduction of the size of the models of a formula, modeling for instance the executions of a Spark job, is one of the possible solutions to reduce the time and the memory needed by the solver to get the result. This section outlines the rationale that is applied to the model of the Spark job, already presented in deliverable D3.6 [1], and that allows such improvement.

As pointed out in D3.6 (Section 4.3), the executions of Spark jobs can be modeled by means of a temporal logic formula written in CLTLoc [8]. The satisfiability problem for CLTLoc is decidable and can be computed through the Bounded Satisfiability Checking (BSC) technique. Solving the saltisfiability problem of a logical formula consists in an assignment of a value to the logical elements that occur in the formula. For instance, given a propositional formula $(A \vee B) \wedge C$, the satisfiability problem for the formula might be the assignment $A = true$ and $C = true$ which makes the overall formula evaluation equal to true.

Time in logic can be represented by positions in the timeline, for instance, through discrete points associated with natural numbers in $\mathbb{N}$. When a temporal logic language is considered, a model is therefore an assignment of values to the logical elements of the formula for all the time positions defining the timeline. A BSC decision procedure looks for a model of the formula that consists of a fixed and finite number of time positions. In general, the less the time positions are in the model, the more the resolution of the formula is feasible in term of time and memory to run the algorithm. The changes implemented in the temporal logic model presented in deliverable D3.6 pursue this objective, which allow the solver to find a "small" solution (i.e., an assignment) for the formula.

5.1.1 CLTLoc Spark job modeling

The rest of the section briefly recalls the main parts of the model of D3.6 and, afterwards, introduces the modifications applied to it. The CLTLoc model makes use of a finite set of atomic propositions and discrete counters to represent a set of feasible job executions. The atomic propositions model the starting, the execution and the termination of the stages in a job and also specify the duration of the tasks and the total number of CPU cores that the stages can use to run the tasks. A trace satisfying the CLTLoc model is a possible schedule, over the time, of the tasks composing the stages of the job; i.e., it represents a possible task ordering which can be executed by means of a finite set of cores.

Each task is executed by one CPU core; therefore, the number of running tasks depends on the number of the available cores. Modeling the execution of tasks requires the following counters whose value vary over the time.

- $\texttt{runTC}_i$ - (*runningTasksCounter*): Number of tasks currently running for stage i;
- $\texttt{remTC}_i$ - (*remainingTasksCounter*): Number of tasks that still have to be executed for stage i;
- $\texttt{avaCC}$ - (*availableCoresCounter*): Number of cores currently available to execute the job.

Finally, the two constant parameters that are specified by the designer to define the verification instance are:

- $\texttt{TOT\_TASKS}_i$ - Total number of tasks needed to complete job i
- $\texttt{TOT\_CORES}$ - Total number of cores available in the cluster

The modification applied to the model mainly affects the formulae modeling the behavior of tasks. Spark tasks are in general executed in parallel as data in Spark can be elaborated by many concurrent tasks of the same type that execute the same functionality on different partitions of the data (having all the same type). Therefore, tasks of the same type are always brought together to form batches, whose size depends on the number of the available CPU cores in the cluster. The behavior of each batch of tasks is modeled by logical formulae defining the beginning, the end and the execution of the batch. The following constraints are modeled through CLTLoc:

- if a batch of tasks starts, i.e., atom $\texttt{startT}_i$ holds, then: (i) the execution of the batch cannot finish at the same time (i.e., $\neg\texttt{endT}_i$ must hold), (ii) in the previous time position, the stage was enabled to run and (iii) a new batch cannot start $\neg\texttt{startT}_i$ until the termination of the current one.
- any execution of a batch of tasks is started with $\texttt{startT}_i$ and ended with $\texttt{endT}_i$, respectively; and that if a batch is running then, at the same time, the corresponding stage is running.
- The termination of a batch of tasks entails that $\neg\texttt{endT}_i$ holds since the position where the current batch was started.

Other CLTLoc formulae determine how counters $\texttt{runTC}_i$ and $\texttt{remTC}_i$ vary their value over the time. Three formulae determine the value of $\texttt{runTC}_i$ and $\texttt{remTC}_i$ during the execution of the batch. In particular,

- the variation of the value of $\texttt{runTC}_i$, between two adjacent time positions, is the sufficient condition to make $\texttt{startT}_i$ or $\texttt{endT}_i$ true. Therefore, between $\texttt{startT}_i$ and $\texttt{endT}_i$, $\texttt{runTC}_i$ cannot vary.
- the variation of $\texttt{remTC}_i$ is the sufficient condition to activate the execution of a batch (i.e., $\texttt{startT}_i$ holds).
- if the execution of a batch of tasks is starting, the number $\texttt{runTC}_i$ of the running tasks in the batch is the difference of the (number of) remaining tasks at the beginning of the batch (i.e., value $\texttt{remTC}_i$) and the remaining tasks in its preceding position.

The duration of the processing phases undertaken by the tasks is represented by means of clocks. Clock $t_{\texttt{phase}_j}$ measures the duration of the $\texttt{runT}_j$ phases for each task j and its value is set to zero every time a batch of tasks starts (i.e., $\texttt{startT}_j$ holds).

The main change applied to the model is related to the following Formula (1) of D3.6 that limits the duration of the execution of a batch of tasks. If there is a batch currently running (i.e., $\texttt{runT}_i$ holds) then $\texttt{runT}_i$ holds until a time position when the value of clock $t_{\texttt{phase}_i}$ is between $\alpha_i \pm \epsilon$ and $\texttt{endT}_i$ is true.

$$\bigwedge_{i\in S}\left(\begin{array}{l}(\texttt{runT}_i \Rightarrow\\ (\texttt{runT}_i \wedge \neg\texttt{endT}_i)\mathbf{U}((\alpha_i-\epsilon \le t_{\texttt{phase}_i} \le \alpha_i+\epsilon) \wedge \texttt{endT}_i)\end{array}\right) \tag{1}$$

The rationale of the new formula replacing Formula (1) is the following. The interval between $\texttt{startT}_i$ and $\texttt{endT}_i$ does not represent the execution of one batch of tasks $\alpha \pm \epsilon$ time units long but it models the executions of (one or) many batches of tasks, whose duration is proportional to the actual number $\texttt{runT}_i$ of running tasks. Modeling the aggregation of batches can be done either by means of parametric formulae or by encoding the size of the batches into the formulae, limiting therefore the possible solution set that the solver should consider while it is solving the satisfiability of the model.

- The parametric modeling lets the number of running tasks $\texttt{runT}_i$ be unconstrained (as in Formula (1)) while it constrains the value of the duration of an aggregation of batches, executed between $\texttt{startT}_i$ and $\texttt{endT}_i$. The total duration $t_{\texttt{phase}_i}$ is set with value $k \cdot \alpha$, where k is a constant that depends on the number of batches in the aggregation. The value k is a integer in the set $\{1, \dots, \frac{\texttt{TOT\_TASKS}_i}{\texttt{TOT\_CORES}}\}$ and indicates how many batches (composed at most of $\texttt{TOT\_CORES}$ and all of the same number of running tasks) might be active between $\texttt{startT}_i$ and $\texttt{endT}_i$. Value $\frac{\texttt{TOT\_TASKS}_i}{\texttt{TOT\_CORES}}$ is the maximum number of batches that must be executed to elaborate all the $\texttt{TOT\_TASKS}_i$ tasks of the i-th stage by means of $\texttt{TOT\_CORES}$ CPUs.

- The non-parametric modeling guides the solver by limiting the sets of values of variable $\texttt{runT}_i$ and, consequently, of the duration of an aggregation of batches (with similar constraints to those of the parametric formula). The value of $\texttt{runT}_i$ belongs to a pre-computed subset of $\{1, \dots, \texttt{TOT\_CORES}\}$ that, for instance, might be composed of all the multiple of 3 from 1 to $\texttt{TOT\_CORES}$.

The following Formula (2) is the parametric version of Formula (1), where n_{rounds} is $\frac{\texttt{TOT\_TASKS}_i}{\texttt{TOT\_CORES}}$ and Formula (3) defines the update of variable $\texttt{remTC}_i$ by an amount that is equal to k times the number of active tasks $\texttt{runTC}_i$.

$$\bigwedge_{i \in S} \Big(\ (\texttt{runT}_i \Rightarrow (\texttt{runT}_i \wedge \neg \texttt{endT}_i) \mathbf{U} (\phi \wedge \texttt{endT}_i) \ \Big) \tag{2}$$

$$\phi := \bigvee_{k=1}^{n_{rounds}} \left(\begin{array}{c} \texttt{remTC}_i = \mathrm{Y}(\texttt{remTC}_i) - k \cdot \texttt{runTC}_i \\ \wedge \\ (k \cdot \alpha_i - \epsilon \leq t_{\texttt{phase}_i} \leq k \cdot \alpha_i + \epsilon) \end{array} \right) \tag{3}$$

The experimental results pointed out that the parametric modeling is, in general, not faster than the original one as the complexity for solving the constraints in Formula (3) turns out to be a relevant part of the resolution of the satisfiability problem. However, the outcome traces produced when the formula is satisfiable show a better scheduling of the tasks over the time than the one produced by the non-parametric version and the one resulting from the old modeling of D3.6. On the other hand, the non-parametric version is, in general, much faster than the other modelings (the parametric and the old one in D3.6) but it sometimes fails as it cannot find a model for the (non-parametric) formula, that is shown to be unsatisfiable, whilst the parametric model can yield a model.

5.2 Model size reduction by means of labeling

Experimental analysis of the execution times registered for verification tasks with the first version of the Spark formal model showed that, by increasing the size of the DAG (i.e., by adding stages), the time needed to perform verification in most cases increases exponentially. As could be expected with this kind of models, this is an instance of the state-space explosion problem. In fact, for each new node in the DAG, a new dedicated set of formulae is added to the model, significantly augmenting the state space of the possible solutions. For this reason, the fewer variables are in the model, the less execution time is affected.

The strategy we adopted to mitigate the problem exploits the fact that, given the dependencies between stages executions, there are stages that will never be executed in parallel. Because of this structural characteristic, a way to reduce the number of variables in the model is to "reuse" as much as possible some variables for representing the stages whose execution is mutually exclusive. For this reason we partition the graph by assigning the same "label" to the stages that can use the same set of variables. In this way the set of variable will be generated for each label instead of for each stage.

A trivial example for explaining the advantage of such approach is the case of a "linear" DAG (Fig. 7), where all the stages have to be executed in sequence and there is no possibility of parallelizing them. In this case, there is always one and only one stage active, therefore it is sufficient to use the formulae of one single stage to represent the execution over time of all of the stages. According to our procedure, only one label ($< 0 >$ in Fig.7) will be assigned to all the nodes in the graph, and only one set of formulae will be generated.

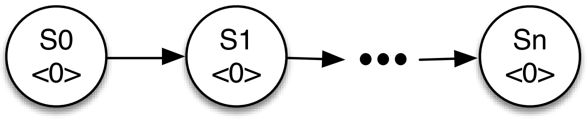


Figure 7: Linear DAG

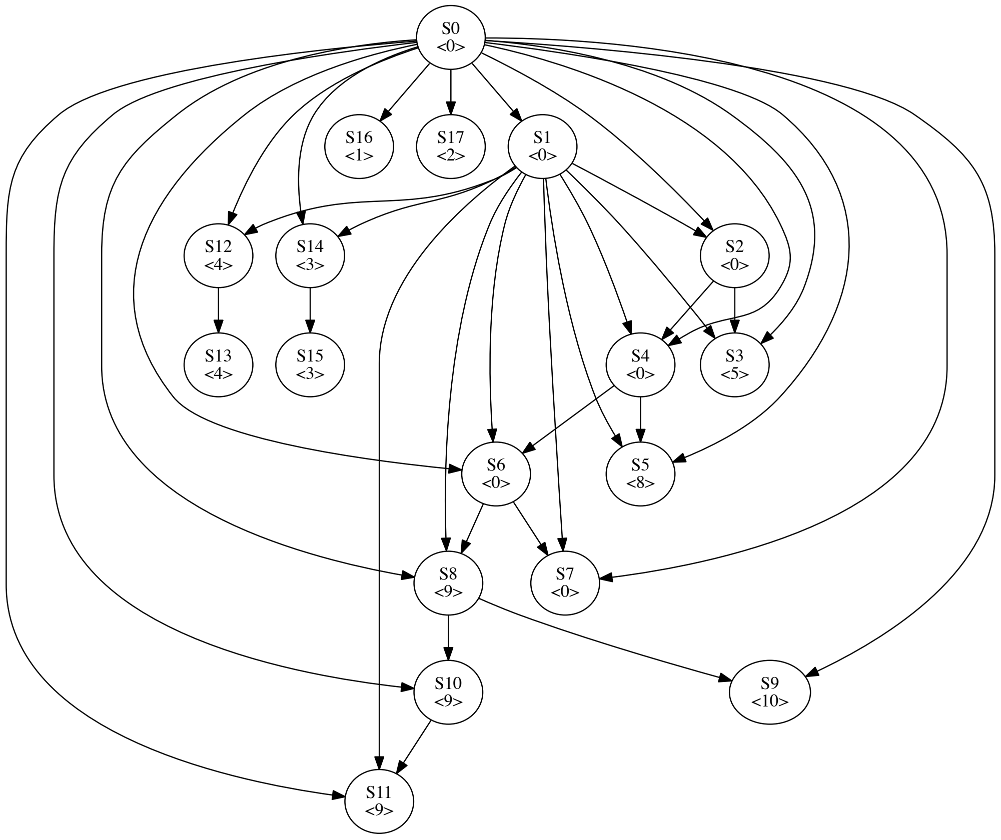


Figure 8: DAG generated for the k-Means application. Labels are represented between angle brackets.

The labeling algorithm performs a breadth-first visit of the graph and assigns labels to all the nodes. Whenever a set of new child nodes is found, the algorithm checks if it is possible to assign them labels already used for the parent nodes. Only when no existing label is available, a brand new label is created and assigned to the node. Figure 8 shows the application of labeling to the DAG generated from a Spark application (implementing an iteration of the K-Means algorithm). The DAG is composed of 18 nodes and the labels that are created (shown between angle brackets in the figure) by the algorithm are 9. This means that, for this particular DAG, it is possible to represent the execution of all the stages in the formal model by generating formulae for only 9 stages instead of 18, halving the size of the final model.

6 Use case

In this section we propose again the use case that we presented in [1], in order to show how, in its final version **D-VerT** supports the verification of Spark applications starting from their UML design.

Listing 2 shows the Python code of the simple application which performs several operations over a text file in which all the lines have the format: *type:name*. The final output is the list of all the name words whose type is different from 'verb' and whose first letter equal to last letter.

Listing 2: Python code representing the example Spark application.

```
from pyspark import SparkContext
sc = SparkContext('local', 'example')
x = sc.textFile("dataset.txt").map(lambda v: v.split(":"))
.map(lambda v: (v[0], [v[1]]))
.reduceByKey(lambda v1, v2: v1 + v2)
.filter(lambda (k,v): k != "verb")
.flatMap(lambda (k, v): v)

y = x.map(lambda x: (x[0], x))
.aggregateByKey(list(),
lambda k,v: k+[v],
lambda v1, v2: v1+v2)

z = x.map(lambda x: (x[-1], x))
.aggregateByKey(list(),
lambda k,v: k+[v],
lambda v1, v2: v1+v2)

result = y.cartesian(z)
.map(lambda ((k1,v1), (k2, v2)):
((k1+k2), list(set(v1) & set(v2))))
.filter(lambda (k,v): len(v) > 1).collect()

print(result)
```

Figure 9 depicts the corresponding UML diagram that is generated through the Papyrus editor in the DICE IDE. Each Opaque Action node corresponds to an operation reported in Listing 2 and is tagged either as a $<< SparkMap >>$ or as a $<< SparkReduce >>$ according to its operation Type. The *MapType* (respectively *ReduceType*) is set for all the nodes and it will be fundamental to determine if new stages need to be created in the execution DAG. For example, the two *aggregateByKey* operations, being two shuffle transformations, will cause the creation of two different stages. The generated execution DAG is shown in Figure 10. It can be noticed how the labeling generates two labels ($< 0 >$ and $< 1 >$) to cover the graph.

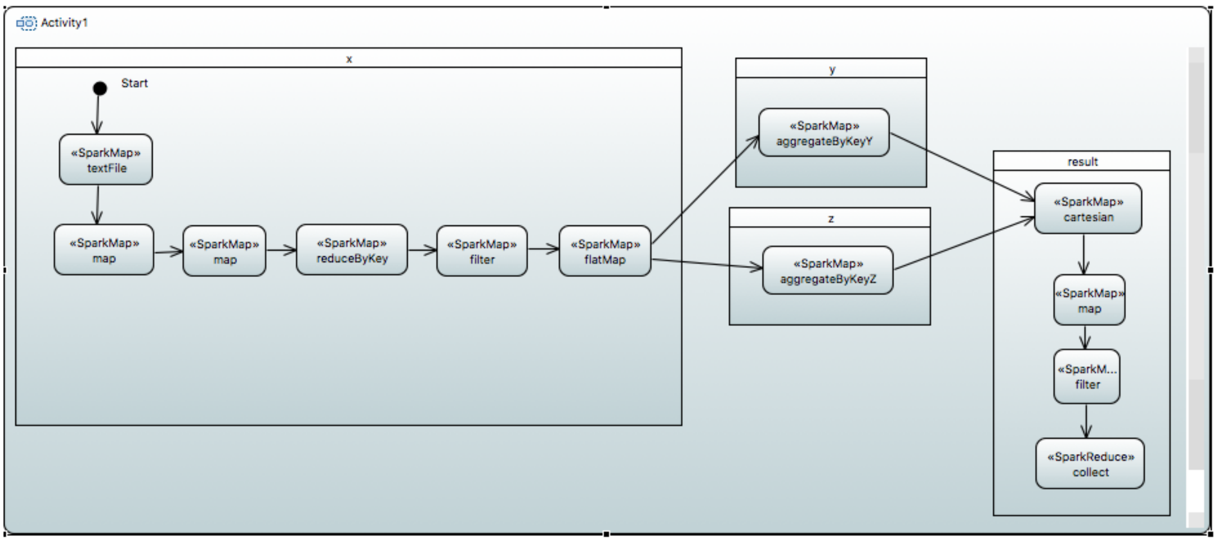


Figure 9: UML Activity diagram of the use case application.

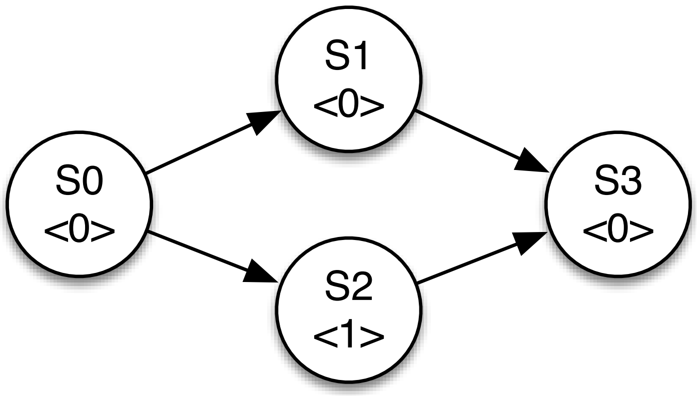


Figure 10: Labeled execution DAG generated from the use case application.

7 Remark on the use of Containerizing technologies

Containerizing technologies do not affect the formal models that were developed to support safety verification in DICE. The two temporal logic models for Storm topologies and Spark job were devised to point out a very specific aspect of the applications implemented with such technologies which is not dependent on the way an application is deployed and distributed over a cluster. The underlying deployment of the nodes executing the application, in fact, does not affect the temporal features of the running application itself, that are related, for instance, to the time needed by a Spark/Storm node to carry out its functionality or to the number of input messages produced by a source node in a Storm topology. For this reason, being the verification modeling completely transparent with respect to the deployment of the application, no extensions to the work achieved so far should be considered to include temporal aspects of containerizing technologies.

8 Conclusions

The primary focus of WP3 - Verification activities was on developing the abstract models supporting safety verification of applications implemented by two baseline data intensive technologies (Apache Storm and Apache Spark) in the DICE research agenda.

We implemented such models by means of a logical approach and we developed them by using CLTLoc logic. We run verification on them by considering standard benchmark applications and we extended existing external tools to let them support the characteristics of the new models.

We designed the models to be configurable and provided a layered structure to facilitate the future integration with the DICE framework by decoupling the verification layer from the DTSM diagrams that the designer exploits to design the application. The activities of the WP3 - Verification have been also focused on the fulfillment of requirements **R3.1** and **R3.2** as they were necessary to integrate **D-VerT** with the DICE IDE. We developed ad-hoc the model-to-model transformations producing verifiable models from the user design, that is realized through the verification toolkit in the DICE IDE.

Beside the temporal logic models, other formalisms were explored to represent Storm applications, opening new possible research activities.

8.1 Upcoming work

In the next months, the activity of the work concerning formal verification in WP3 will focus on bug-fixing and code-tuning of **D-VerT** with the collaboration of DICE industrial partners to support them in the demonstration of the technology.

Requirement ID	Description	Coverage	To do
R3.1	M2M Transformation	100 %	-
R3.2	Taking into account relevant annotations	100 %	-
R3.3	Transformation Rules	100 %	-
R3.7	Generation of traces from system model	100 %	-
R3.10	SLA specification and compliance	100 %	-
R3.12	Modelling abstract level	100 %	-
R3.15	Verification of temporal safety/privacy properties	90 %	-
R3IDE.2	Timeout Specification	100 %	-
R3IDE.4.2	Loading the properties to be verified	100 %	-
R3IDE.5	Graphical output	100 %	-
R3IDE.5.1	Graphical output of erroneous behaviours	100 %	-

Table 1: Requirement coverage at month M30. R3.15 is coverd or 90% as the privacy concern requirements have been moved to Trace-checking analysis in Work Package 4 (details about privacy are supplied by deliverable D4.4 and D2.4).

References

[1] M. Bersani et al. *DICE Verification Tool - Intermediate Version*. Tech. rep. www.dice-h2020.eu. DICE Consortium, 2017.

[2] The DICE Consortium. *Requirement Specification*. Tech. rep. available from www.dice-h2020.eu. European Union's Horizon 2020 research and innovation programme, 2015.

[3] The DICE Consortium. *Requirement Specification - Companion Document*. Tech. rep. available from www.dice-h2020.eu. European Union's Horizon 2020 research and innovation programme, 2015.

[4] The DICE Consortium. *Design and Quality Abstractions - Final 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, 2017.

[5] D. Perez et al. *DICE simulation tools - Final version*. Tech. rep. to appear. DICE Consortium, 2017.

[6] M. Gil et al. *DICE Framework – Initial version*. Tech. rep. available from www.dice-h2020.eu. DICE Consortium, 2016.

[7] D. Ardagna et al. *DICE Transformations to analysis models*. Tech. rep. www.dice-h2020.eu. DICE Consortium, 2016.

[8] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. "A Tool for Deciding the Satisfiability of Continuous-time Metric Temporal Logic". In: *Proc. of TIME*. 2013, pp. 99–106. DOI: `10.119/TIME.2013.20`.