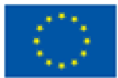
Ref. Ares(2017)531483 - 31/01/2017

Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE Verification Tools - Intermediate Version

Deliverable 3.6

Deliverable:	D3.6
Title:	Verification Tools - Intermediate Version
Editor(s):	Marcello M. Bersani
Contributor(s):	Madalina Erascu, Francesco Marconi, Matteo Rossi
Reviewers:	
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-Jan-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 presents the status of the activities related to task T3.3, which concerns safety verification of data-intensive applications (DIA). The temporal aspects of the runtime behavior of DIAs are the focus of verification in DICE, which, therefore, is founded on temporal formalisms allowing analysis of properties over time. To this end, temporal logics are extensively adopted in this context, modeling both the DIAs and the properties to verify. Further details on Task3.3 can be found in D1.1 - State of the art analysis, and D1.2 - Requirements specifications.

Deliverable D3.6 is the second of three deliverables (D3.5, D3.6, and D3.7) reporting the status of the development activities of the DICE Verification Tool (**D-VerT**). **D-VerT** allows application designers to evaluate their design against safety properties, such as reachability of undesired configurations of the system, meeting of deadlines, and so on.

Deliverable D3.6 describes the extension of **D-VerT** that allows developers to carry out verification of Spark applications. The document briefly reports on the integration of **D-VerT** in the DICE IDE and details the temporal logic model devised for Spark applications, presented in deliverable D3.2 – Transformations to Analysis models. The verification approach adopted for Spark applications is the same as the one used for the analysis of Storm topologies although the class of formulae that we use to represent Spark job is different from the class previously used for Storm. However, in both the cases, verification is based on satisfiability checking of temporal formulae. **D-VerT** relies on a unique verification engine solving the satisfiability problem, that is the **Zot**[1] tool. The novelty of the contribution, described in this document, consists of the modeling of the executions of a Spark job and the definition of a safety property of interest for Spark developers.

Moreover, deliverable D3.6 shows the results of the research activity related to the study of verification approaches and models that are different from those based on temporal logic verification. The analysis proposed in this document is the conclusion of the preliminary results that are reported in D3.5 and that were obtained from the first investigation on the verification of systems specified in a first-order logic (FOL) ended with the theory of arrays.

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

Copyright © 2017, DICE consortium – All rights reserved

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

Copyright © 2017, DICE consortium – All rights reserved

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 . . . 7
- 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 Verification tool overview . . . 13

4 Modeling Spark applications . . . 14
- 4.1 Terminology . . . 15
- 4.2 Modeling assumptions and Job model . . . 15
- 4.3 Temporal Logic Model of Spark Jobs . . . 16
  - 4.3.1 Stage-specific Atomic Propositions . . . 16
  - 4.3.2 Task-specific Atomic Propositions . . . 17
- 4.4 Temporal logic model . . . 17
  - 4.4.1 Stage formulae . . . 17
  - 4.4.2 Tasks formulae . . . 18
  - 4.4.3 Resource Constraints . . . 18
  - 4.4.4 Counters Formulae . . . 18
  - 4.4.5 Initialization . . . 19
  - 4.4.6 Clocks Formulae . . . 19

5 Temporal analysis of Spark applications . . . 20

6 Support for Spark in the Verification Tool . . . 22

7 Validation . . . 24

8 Modeling and Verifying Storm Applications . . . 27

9 Conclusion and future works . . . 33
- 9.1 Further work . . . 33

References . . . 35

A Details of the Formal Models . . . 36
- A.1 First Order Logic Model. Example 1 . . . 36
- A.2 First Order Logic Model. Example 2 . . . 37

Copyright © 2017, DICE consortium – All rights reserved

List of Figures

1 Sequence diagram of the interaction between the user and the components in the DICE framework. . . . 13
2 Simplified Component Diagram showing the main modules of **D-VerT**. . . . 22
3 Activity Diagram representing the workflow from the UML design of the DIA to the execution of the verification task. . . . 23
4 Execution DAG generated to run the application. . . . 24
5 Graphical representation of the output trace produced by **D-VerT**. . . . 26
6 Finite automata describing the states of a spout/bolt 27
7 Topology Example 28

List of Tables

1 Requirement coverage at month 24. 33

List of Listings

1 Python code representing the toy Spark application. 24
2 Simplified JSON file describing the execution DAG for the example application. 25

1 Introduction

The tool for verification of safety aspects of data-intensive application is the *DICE Verification Tool* (**D-VerT**). The definition of the tool requirements and the functionality were already presented in the first document – Deliverable D3.5 – related to verification and Task 3.3.

D-VerT allows application designers to evaluate their design against relevant temporal properties, in particular safety ones (such as reachability of undesired configurations of the system, meeting of deadlines, and so on). The outcome of the verification is used for refining the model of the application at design time, in case anomalies are detected by **D-VerT**. In its initial version, **D-VerT** already supported verification of Storm topologies. Deliverable D3.5 presents the verification workflow and the intermediate input format for the tool that allows for decoupling the DICE platform from the verification engine. Deliverable D3.1 describes the integration of **D-VerT** in the DICE IDE and shows how the DICE developer can define the Storm application undergoing verification in the DICE framework, at the DTSM level of the design workflow. In short, **D-VerT** reads a DTSM diagram defining the application undergoing verification and translates it into a JSON file which is then elaborated by the verification engine along with the property to be checked. In case a counterexample is found, the tool shows to the user the execution trace that violates the property.

This document describes the intermediate version of **D-VerT**, which is extended with new functionalities tackling the verification problem for Spark applications. To this end, we devise 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). In particular, the verification problem consists in determining the existence of executions that violate specific temporal properties. For example, two complementary analysis can be performed: *feasibility analysis* and *boundedness analysis*. The former aims at checking if there exist an execution of the system whose duration is lower than a specific deadline, witnessing the feasibility of such execution, while the latter checks—making strong assumptions on the idle time of the application—whether all the possible executions of the system are below a certain threshold (this corresponds to verifying if exists one execution which takes longer than the deadline).

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.

- Task3.3 intends to develop a verification framework that enables the automatic evaluation of safety properties of DIAs, limited to Storm topologies and Spark jobs. DIA models are validated by means of model checking with the aim of verifying temporal aspects of the application behavior.
- Verification is carried out through satisfiability checking of Constraint LTL over-clocks (CLTLoc) formulae, that represent (an abstraction of) the DIA behavior over time. By means of a graph-based representation of the DIA and a set of user-defined non-functional parameters, the DICE designer can perform the safety analysis at the DTSM level of the DICE design workflow. To verify the

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

Copyright © 2017, DICE consortium – All rights reserved

application, the DTSM diagram is then translated by **D-VerT** into a CLTLoc formula and verified by the model checker.

- The analyses (i), (ii) and (iii) above can be performed separately or together. In the second case, typically one runs verification first and gets feedback if the safety property holds or not. Depending on the result, verification is ran again with new input or simulation and then optimization is performed.
- The outcome of the verification task allows us to draw conclusions whether the properties are satisfied, and not where the problem occurred and how to fix it. However, in the case that the property is not satisfied the output trace gives a hint to the designer on what should be fixed.

The work undertaken in the last year has been focused on the following activities, all fitting the goal of Task T3.3.

1. *Definition and implementation of a temporal model for Spark jobs*. The activity provided **D-VerT** with a new functionality that enables the temporal assessment of Spark jobs. The result of this work constitutes the main argument of the document and it is described in Section 4.
2. *Integration of **D-VerT** in the DICE framework*. The activity fulfills the requirements R3.1, R3.2, R3.7, R3.12, R3.15, R3IDE.4.1, R3IDE.4.2, R3IDE.5 and R3IDE.5.1. **D-VerT** can be used through the DICE Eclipse environment as it has been fully integrated in the DICE framework since month 18. Some preliminary results on the integration of **D-VerT** in the DICE framework were reported in deliverable D3.1 - Transformation to Analysis Models [**d3.1**].
3. *Modeling and verification of Storm applications with First-Order Logic with arrays*. This activity investigated the use of infinite-state verification techniques for the safety assessment of DIAs, to complement the main temporal logic-based approach. The result of this work is described in Section 8.

1.2 Motivation

The analysis of correctness is fundamental to produce systems whose behavior at runtime is guaranteed to be correct. However, the notion of correctness is general and needs to be refined to suit a given scenario. Appropriate criteria have to be considered, according to the kind of system that is taken into account and to the user requirements that the implemented system should exhibit at runtime. Verification in DICE aims to define the meaning of correctness for DIAs and provides implementation of tools supporting formal analysis of DIAs. Task3.3 is motivated by this need and promotes safety verification of DIAs through the use of **D-VerT**.

Safety verification in DICE is performed to check, in the DIA model, the reachability of bad configurations, i.e., a malfunction which consists of behaviors that do not conform to some non-functional requirements specified by the QA_ENGINEER. Task T3.3 focuses on the analysis of the effect of an incorrect design of timing constraints which might cause a latency in the processing of data. This aspect is quite relevant in applications such as those considered in DICE. The unforeseen delay can actually lead the system to incorrect behaviors that might appear at runtime in various form depending on the kind of application under development. For instance, in a streaming application, the delay might cause accumulation of messages in the node queues and lead possibly to memory saturation, if no procedures dealing with the anomaly take action. In a batch application, an unpredicted delay might affect the total processing time and alter the application behavior which, in turn, violates the SLA with the clients.

The verification process 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. It is designed to be carried out in an agile way: the DIA designer performs verification by using a lightweight approach. More precisely, **D-VerT** fosters an approach whereby formal verification of DIAs is launched through interfaces that hide the complexity of the underlying models and engines. These interfaces allow the user

Copyright © 2017, DICE consortium – All rights reserved

to easily produce the formal model to be verified along with the properties to be checked and eliminates the need for experts of the formal verification techniques.

1.3 Structure of the deliverable

Section 1 provides an overview on the objectives and the motivation of Task3.3; Section 2 summarizes the fundamental requirements that Task3.3 has to fulfill. Section 3 outlines the verification workflow that **D-VerT** employs. Section 4 elaborates on the behavior of Spark at runtime, shows the assumptions that allowed the definition of the temporal logic model of Spark jobs and details all the formulae that model an execution of a job. Section 6 describes how the verification is carried out in **D-VerT** and shows the interaction between its components. Section 7 shows an example of verification of a Spark job. Section 8 elaborates on the advancements in the Storm model based on First-order logic and provides an analysis of the achievements. Section 9 draws some conclusions on the deliverable. Appendix A provides all details of the First-order logic model of Storm topologies.

Copyright © 2017, DICE consortium – All rights reserved

2 Requirements and usage scenarios

Deliverable D1.2 [1, 2] 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, these requirements and use case scenarios and explains how they have been fulfilled in the current **D-VerT** prototype.

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 [1, 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 [1] are the following:

Copyright © 2017, DICE consortium – All rights reserved

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

Copyright © 2017, DICE consortium – All rights reserved

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

Copyright © 2017, DICE consortium – All rights reserved

3 Verification tool overview

D-VerT (DICE Verification Tool) is the verification tool integrated in the DICE framework. Verification is performed on annotated DTSM models of Storm or Spark applications that contain suitable information about the timing features of the application undergoing analysis. The tool has a client server architecture, in which the client component is an Eclipse plugin that is fully integrated with the DICE IDE and the server component is a RESTful web server. The current version of **D-VerT** supports queue saturation analysis for Storm topologies (already presented in [**d3.5**]) and partially supports two kind of timing analyses (feasibility and boundedness) for Spark jobs, which are presented in this document. With partial support we mean that currently only the server component supports the verification of Spark applications (as described in Section 6), while the extension of the client component is still under development.
The verification process consists of the following steps. The DICE designer draws a class diagram in the DICE IDE representing the DTSM model of the DIA and, afterwards, provides all the annotations required for the analysis, based on the selected technology that he/she employs to implement the application. When the user starts the analysis, the annotated DTSM model is converted into a formal model that represents the abstracted behavior of the application at runtime. Based on the kind of system to implement (i.e., Storm or Spark), the tool selects the class of properties to verify and performs the analysis. Finally, when the outcome is available, the user requests **D-VerT** to show the result in the DICE IDE to see if the property is fulfilled or not and, in the negative case, the trace of the system that violates it.
The client component manages the transformation from the DTSM model defined by the user to an intermediate JSON object which is then used to invoke the server. The server component, based on the content of the JSON file generates the formal model which is then fed to the core satisfiability/model checking tool. This tool is in charge of verifying if the property holds for provided set of formulae (formal model of the system).

The sequence diagram in Figure 1 shows the set of interactions taking place among the different components whenever the user creates a DTSM model, launches the verification of a specific model and asks for the results of the verification. More details about the **D-VerT** components and the integration of the plugin in the DICE framework can be found in [**d3.1**] and [**d3.5**].

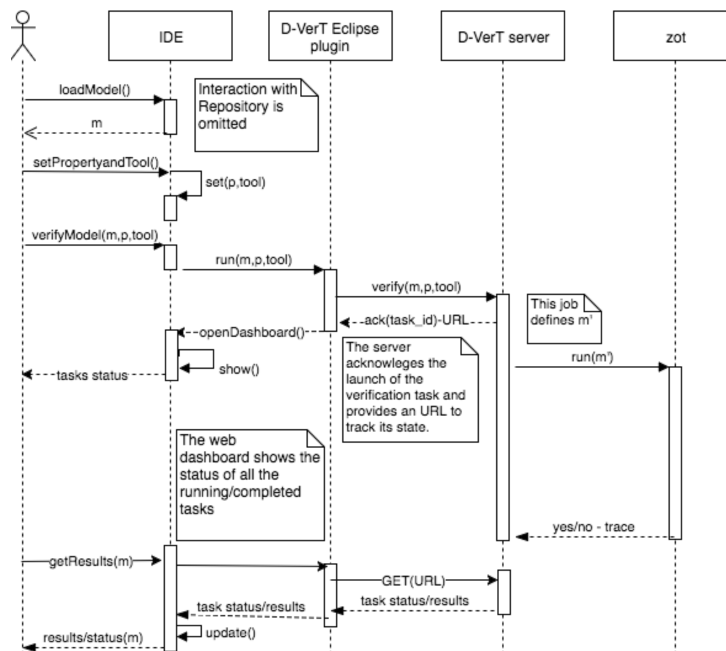


Figure 1: Sequence diagram of the interaction between the user and the components in the DICE framework.

Copyright © 2017, DICE consortium – All rights reserved

4 Modeling Spark applications

This section is related to the work that was done on the verification of Spark applications and focuses on the results that were obtained during the second year of DICE.

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 physical infrastructure which executes a Spark application can vary from one virtual machine to clusters of thousands of nodes that are managed transparently by the Spark manager. Spark offers a lightweight development environment consisting of a rich set of APIs. This includes a variety of high-level operators for in-memory parallel data manipulation, such as map, reduce or filtering functions, as well as operators for data persistence. The computational model of Spark is specifically designed to guarantee data parallelism and fault-tolerant executions. Data are uniformly partitioned across different nodes, and multiple partitions can be concurrently processed by applying the same operations in parallel. Partial results are then combined to obtain more complex data through aggregating functions. The fundamental data structure in Spark is the so-called Resilient Distributed Dataset (RDD) that is a read-only multiset of data items distributed over a cluster of machines and maintained in a fault-tolerant way. An RDD can contain any object type and is created by loading an external dataset or by distributing a collection of objects generated by the application. RDDs support two types of operations:

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

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 operations executed in parallel by scheduling their operations in a proper way. It keeps track of the dataset that the transformation operates and computes the transformations only when an action is called. Fault-tolerance is achieved by caching the sequence of operations that produced each RDD so that any RDD can be reconstructed in the case of data loss.

Spark computations can be represented with a directed acyclic graph (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. A stage can be executed only if all its predecessors have finished their computation.

Jobs and stages are logical notions representing the computation of Spark applications. At runtime, the actual computation is realized through workers, executors and tasks over a cluster of nodes. A *worker* is a node that can run application code in the cluster and that contains some executors, i.e., processes that run tasks and possibly store data on a worker node. The *Driver program* is the process running the main() function of the application and that creates the *Spark Context*, that is the object representing a running instance of a Spark application. The Driver contains the DAG scheduler and the application code and manages the execution of the processes in the cluster with the aid of a *cluster manager*, which allocates resources across the running applications. To run a Spark job, the Driver first acquires the executors on nodes in the cluster by means of the cluster manager. Then, Spark distributes the application code (passed to SparkContext) to the executors and, finally, Spark Context sends the tasks to the executors, activating the computation.

Spark execution is done as follows:

- Spark first creates the operator graph based on the code submitted to the Spark intepreter.

Copyright © 2017, DICE consortium – All rights reserved

- When an action has to be executed, the operator graph is submitted to a DAG Scheduler, which identifies the stages. The DAG scheduler arranges the operators and produces the stage graph.
- The stages are then passed to the Task Scheduler which launches tasks via the cluster manager.
- The workers execute the tasks defining the stages of the job.

4.1 Terminology

The main concepts that are involved in the logical model shown in the next Sect. 4.2 are the following.

Job : The Spark user application.

Stage : sequence of operations on data that can be executed in parallel and that produces a result which enables the execution of operations belonging to following stages.

Task : Unit of computation contributing to the definition of a stage. A task only operates on a single data partition and its execution is managed by one executor.

DAG : Directed Acyclic Graph of stages manipulating RDDs. This graph can be derived from the application code by examining the dependencies among the operations on the RDDs.

Executor : The process executing a task.

Driver : The process running the Job over the Spark Engine.

4.2 Modeling assumptions and Job model

As already pointed out in the introduction, **D-VerT** helps the DIA designer in doing timed analysis of Spark jobs. The analysis 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, and with the number of CPU cores executing the task. The number of cores associated with a task depends on the number of data partitions that the task elaborates. The data partitions are obtained by the Spark engine when the Spark job is launched and it is based on the dimension of the input data, as the partition size is a parameter that can be set through the Spark Context. 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. The task latencies are supposed to be obtained by application profiling and must be known before performing the analysis. The functional aspects implemented in the source code are not taken into account. Verification concerns non-functional properties of the DIAs and the task functionalities are modeled with their timing requirements.

This section describes in detail the CLTLoc formulae that models the runtime behavior of a Spark job given its DAG of stages and tasks. However, it does not focus on the procedure that one can use to extract the RDD dependency graph from the user code or to derive the DAG of stages from the dependencies among the RDD implemented in the source code. This functionality will be integrated in the forthcoming releases of **D-VerT** and it will be described in the final deliverable of Task3.3. The assumptions that are considered in the modeling are described in the following.

- The runtime environment that Spark instruments to run a job is not considered in the modeling. In particular, the cluster manager, the Spark Context, the workers and the executors are not taken into account in the formulae.

Copyright © 2017, DICE consortium – All rights reserved

- The latency generated by the execution of services managing the jobs is considered negligible with respect to the total time for executing the application.

The next two assumptions concern the cluster environment encompassing the Spark job that the formulae represent. They limit the effects that are induced by executing several jobs in the cluster and allows us to devise a logical model which can ignore potential latencies that are caused by concurrent applications.

- 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.

In addition, the following assumptions are specifically related to the Spark job and to the computational resources that are assigned to the processes executing the tasks.

- 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.

4.3 Temporal Logic Model of Spark Jobs

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 are used to model the starting, the execution and the termination of the stages in a job; whereas counters are positive integer variables that keep track of the number of CPU cores that are allocated to run the active tasks. The evolution of the atoms and counters over time describes one, among many, possible computation of the job that fulfills the constraints expressed by the CLTLoc formulae. In particular, the CLTLoc model specifies the duration of the tasks, the total number of CPU cores that the application can use and the precedence relation among the tasks and the stages in order to model job executions that conform to the following requirements.

- The total number of cores that are assigned to the active tasks, at any moment, is less than, or equal to, the total number of cores that are allocated for executing the job.
- The total duration of any task is bounded.
- A stage can start only if:
  - all its predecessors (determined by the DAG) have already completed their execution;
  - there is at least one free core that can be assigned to execute a task of the stage.

A trace satisfying the CLTLoc model is actually 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.

The atomic propositions and counters that are used in the CLTLoc model are described hereafter.

4.3.1 Stage-specific Atomic Propositions

$\texttt{runS}_i$: stage i is running. This phase includes the starting and the termination time instances.

$\texttt{startS}_i$: stage i is starting the execution.

$\texttt{endS}_i$: stage i has terminated the execution.

Copyright © 2017, DICE consortium – All rights reserved

$\texttt{completedS}_i$: stage i terminated the execution. It holds after $\texttt{endS}_i$.

$\texttt{enabledS}_i$: stage i can be executed as soon as there are available CPU cores. Atom $\texttt{enabledS}_i$ holds if all the stages that precede i in the DAG have been completed, even if no cores can be allocated to execute the tasks of i.

4.3.2 Task-specific Atomic Propositions

The atoms $\texttt{runT}_i$, $\texttt{startT}_i$ and $\texttt{endT}_i$ are used to model the behavior of tasks, similarly to $\texttt{runS}_i$, $\texttt{startS}_i$ and $\texttt{endS}_i$ for the stages. For instance, $\texttt{runT}_i$ holds when there are *some* tasks of stage i currently running in the cluster. However, the tasks composing a stage are not individually represented in the CLTLoc model: no atomic propositions express that task j of stage i is running. The model represents the execution of a batch of tasks with no possibility of distinguishing one task from the other.

Each task is executed by one CPU core; therefore, the number of running tasks depends on the number of available cores. Modeling the execution of tasks requires the following counters that are used in the formulae to express that the sum of the number of running tasks in any moment does not exceed the number of cores for the job.

- $\texttt{runTC}_i$ - (*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;
- 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
- TOT_CORES - Total number of cores available in the cluster

4.4 Temporal logic model

A Spark execution DAG is a directed acyclic graph $\mathbf{G} = \{\mathbf{S}, Dep_{i,j}\}$, whose nodes $s_1, s_2, ...s_n \in \mathbf{S}$ are the stages of the Spark job, while the edges $Dep_{i,j}|i,j \in \mathbf{S}$ define dependencies and precedence between stages. $Dep_{i,j}$ means that stage i *depends on* stage j, that is, in order for stage i to be executed, stage j must be completed.

The discrete variable $\texttt{runTC}_i$ ("running tasks counter") represents the number of tasks currently executed for the stage i. $\texttt{remTC}_i$ ("remaining tasks counter") keeps track of the quantity of tasks that still have to run. $\texttt{remTC}_i$ is initialized as $\texttt{TOT_TASKS}_i$ and is decremented by $\texttt{runTC}_i$ every time a batch of $\texttt{runTC}_i$ tasks is completed.

Let orig be a shorthand for $\neg\mathbf{Y}(\top)$ which mark the origin of the trace as it is true only at the first position of the trace.

4.4.1 Stage formulae

A stage i can be either running ($\texttt{runS}_i$ holds) or not running; the latter case happens either when the stage has been already executed or as soon as it will start the computation, namely, when its predecessors complete their tasks and there are available CPU cores. A stage is activated, i.e., $\texttt{startS}_i$ holds, when there is at least one task that starts the execution and no task has been executed so far. If no tasks were executed then the number of the tasks to be processed, represented by variable $\texttt{remTC}_i$, is equal to the number of tasks that the stage has to elaborate ($\texttt{TOT_TASKS}_i$). This situation is modeled through the following Formula 1.

$$\bigwedge_{i \in \mathbf{S}} (\texttt{startT}_i \wedge \mathbf{Y}(\texttt{remTC}_i = \texttt{TOT_TASKS}_i) \iff \texttt{startS}_i) \tag{1}$$

Copyright © 2017, DICE consortium – All rights reserved

A stage terminates, i.e., $\texttt{endS}_i$ holds, when there are no more tasks to be processed, i.e., when $\texttt{remTC}_i$ is equal to 0. Next Formula 2 defines $\texttt{endS}_i$.

$$\bigwedge_{i \in \mathbf{S}} (\texttt{endT}_i \wedge \texttt{remTC}_i = 0 \iff \texttt{endS}_i) \tag{2}$$

The following formulae define the meaning of $\texttt{enabledS}_i$ and $\texttt{completedS}_i$. A stage is completed (i.e., $\texttt{completedS}_i$ holds) when it has been terminated in the past (there is a position before the current one where $\texttt{endS}_i$ held) and a stage i is enabled (i.e., $\texttt{enabledS}_i$ holds) when all the predecessor stages j, such that $Dep(i, j)$ is defined, have been completed.

$$\bigwedge_{i \in \mathbf{S}} (\texttt{completedS}_i \iff \mathbf{P}(\texttt{endS}_i)) \tag{3}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{enabledS}_i \iff \bigwedge_{\substack{j \in \mathbf{S}: \\ Dep(i,j)}} \texttt{completedS}_j) \tag{4}$$

4.4.2 Tasks formulae

The behaviour of each batch of tasks is defined by the following formulae. Formula 5 specifies the necessary conditions that must be true when a batch of tasks starts. When $\texttt{startT}_i$ holds then (i) the execution cannot be finished 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.

$$\bigwedge_{i \in \mathbf{S}} (\texttt{startT}_i \Rightarrow \texttt{runT}_i \wedge \neg\texttt{endT}_i \wedge \mathbf{Y}(\texttt{enabledS}_i) \wedge \mathbf{X}(\texttt{endT}_i \mathbf{R} \neg\texttt{startT}_i)) \tag{5}$$

Formula 6 imposes that 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.

$$\bigwedge_{i \in \mathbf{S}} (\texttt{runT}_i \Rightarrow \texttt{runS}_i \wedge (\texttt{runT}_i \mathbf{S}\, \texttt{startT}_i) \wedge (\texttt{runT}_i \mathbf{U}\, \texttt{endT}_i)) \tag{6}$$

Similarly to Formula 5, Formula 7 defines the necessary conditions so that $\texttt{endT}_i$ holds. The termination of a batch of tasks imposes that $\neg\texttt{endT}_i$ holds since the position where the current batch was started.

$$\bigwedge_{i \in \mathbf{S}} (\texttt{endT}_i \Rightarrow \texttt{runT}_i \wedge \mathbf{Y}(\neg\texttt{endT}_i \mathbf{S}(\texttt{orig} \vee \texttt{startT}_i))) \tag{7}$$

4.4.3 Resource Constraints

The following formula enforces the consistency constraints limiting the number of cores that are allocated to execute the active tasks. In particular, the sum of the number of idle and allocated cores is always equal to TOT_CORES, the number of cores that is assigned to the job.

$$\sum_{i \in \mathbf{S}} (\texttt{runTC}_i) + \texttt{avaCC} = \texttt{TOT_CORES} \tag{8}$$

4.4.4 Counters Formulae

Counter values determine the evolution of the tasks that are executed within the stage. Therefore, their value is always non-negative as they represent positive quantities and the number of the remaining tasks of a stage decreases during its execution. Formula 9 enforces non-negativeness of counters and Formula 10 imposes that the next value of $\texttt{remTC}_i$ (written $\texttt{XremTC}_i$) is less than the value of $\texttt{remTC}_i$ in the current position.

$$\texttt{avaCC} \geq 0 \wedge \bigwedge_{i \in \mathbf{S}} (\texttt{runTC}_i \geq 0 \wedge \texttt{runTC}_i \geq 0) \tag{9}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{remTC}_i \geq \texttt{XremTC}_i) \tag{10}$$

Copyright © 2017, DICE consortium – All rights reserved

The following formulae link the truth value of the events $\texttt{startT}_i$ and $\texttt{endT}_i$ with the value of counters $\texttt{runTC}_i$ and $\texttt{remTC}_i$. Formula 11 correlates variable $\texttt{runTC}_i$ with proposition $\texttt{runT}$ by imposing that a batch is running (i.e., $\texttt{runT}$ holds) when the value of $\texttt{runTC}_i$ of active tasks is strictly positive. The two formulae (12) and (13) determine the value of $\texttt{runTC}_i$ and $\texttt{remTC}_i$ during the execution of the batch. Since the model is not designed to represent core re-balancing operations, the formulae enforce a variation of $\texttt{runTC}_i$ or $\texttt{remTC}_i$ to occur when a batch starts or terminates. In particular, Formula (12) imposes that a variation of the value of $\texttt{runTC}_i$, between two adjacent 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. Similarly, Formula (13) imposes that a variation of the value of $\texttt{remTC}_i$ is the sufficient condition to activate the execution of a batch (i.e., $\texttt{startT}_i$ holds). Finally, Formula 14 defines the relation between the variables $\texttt{runTC}_i$ and $\texttt{remTC}_i$. It states that, 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 (i.e., value $\mathrm{Y}\texttt{remTC}_i$).

$$\bigwedge_{i\in\mathbf{S}} (\texttt{runT}_i \Leftrightarrow \texttt{runTC}_i > 0) \tag{11}$$

$$\bigwedge_{i\in\mathbf{S}} ((\texttt{runTC}_i \neq \mathrm{X}\texttt{runTC}_i) \Rightarrow (\mathbf{X}(\texttt{startT}_i) \vee \texttt{endT}_i)) \tag{12}$$

$$\bigwedge_{i\in\mathbf{S}} (\texttt{remTC}_i \neq \mathrm{X}\texttt{remTC}_i \Rightarrow \mathbf{X}(\texttt{startT}_i)) \tag{13}$$

$$\bigwedge_{i\in\mathbf{S}} (\texttt{startT}_i \Rightarrow (\texttt{runTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{remTC}_i)) \tag{14}$$

4.4.5 Initialization

The initial condition of any Spark job is determined by the following formulae (15)-(17):

- no tasks are running in the origin (Formula 15),
- for all the stages, the number of the remaining tasks is $\texttt{TOT_TASKS}_i$, that is the total number of tasks to be processed by stage i (Formula 16),
- the number of available cores $\texttt{avaCC}$ is the total number of the cores $\texttt{TOT_CORES}$ (Formula 17).

$$\bigwedge_{i\in\mathbf{S}} (\neg\texttt{runT}_i) \tag{15}$$

$$\bigwedge_{i\in\mathbf{S}} (\texttt{remTC}_i = \texttt{TOT_TASKS}_i) \wedge (\texttt{runTC}_i = 0)) \tag{16}$$

$$(\texttt{avaCC} = \texttt{TOT_CORES}) \tag{17}$$

4.4.6 Clocks Formulae

To represent the duration of the various processing phases undertaken by the tasks, different clocks are considered in the model:

- $t_{\texttt{phase}_j}$ measures the duration of the $\texttt{runT}_j$ phases for each task j.
- $\texttt{clock}_{\texttt{idleCores}}$ measures the duration of the $\texttt{idleCores}$ global state, i.e. the state in which some cores are not used while there are runnable tasks.

The following formulae define the reset conditions for the clocks.

- $t_{\texttt{phase}_j}$ **for tasks** - clock resets every time a new batch of tasks starts running.

$$\bigwedge_{j\in S} \Big((t_{\texttt{phase}_j} = 0) \iff (\texttt{orig} \vee \texttt{startT}_j)\Big) \tag{18}$$

(19)

Copyright © 2017, DICE consortium – All rights reserved

- $\texttt{clock}_{\texttt{idleCores}}$ - resets every time some cores become idle.

$$\left((\texttt{clock}_{\texttt{idleCores}} = 0) \iff (\texttt{idleCores} \land \neg \mathbf{Y}(\texttt{idleCores})) \lor \bigvee_{i \in S} (\texttt{startT}_i) \right) \tag{20}$$

Formula 21 limits the duration of the execution of a batch of tasks by imposing that the termination of the batch occurs when the value of clock $t_{\texttt{phase}_i}$ is between $\alpha_i \pm \epsilon$, where α_i is the latency that is obtained by the application profiling and ϵ is a constant defining the delay for processing a data partition by means of tasks of stage i. 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.

$$\bigwedge_{i \in S} \left(\begin{array}{l} (\texttt{runT}_i \Rightarrow \\ (\texttt{runT}_i \land \neg \texttt{endT}_i) \mathbf{U} ((t_{\texttt{phase}_i} \geq \alpha_i - \epsilon) \land (t_{\texttt{phase}_i} \leq \alpha_i + \epsilon) \land \texttt{endT}_i) \end{array} \right) \tag{21}$$

Formula 21 limits the duration of the idleness phases occurring during the execution of the jobs, which happens when no cores are active. This way, the model represents realistic executions of Spark. The Spark engine, in fact, attempts to reduce the number of inactive cores so that the number of running tasks is maximized. To reduce latencies that are caused by inactivity, Formula 21 limits the duration of the idleness phase by imposing that clock $\texttt{clock}_{\texttt{idleCores}}$, that keeps the elapsed time when $\texttt{idleCores}$ is true, is less or at most equal to $\texttt{MAX_IDLE_TIME}_\texttt{j}$.

$$\left(\begin{array}{l} \texttt{idleCores} \Rightarrow \\ (\texttt{idleCores} \mathbf{U} \left((\texttt{clock}_{\texttt{idleCores}} \leq \texttt{MAX_IDLE_TIME}_\texttt{j}) \land \bigvee_{i \in S} (\texttt{startT}_i) \right) \end{array} \right) \tag{22}$$

5 Temporal analysis of Spark applications

Two complementary properties are considered in this deliverable for the temporal analysis of Spark jobs. Before describing them, let S be the set of the stages of the job undergoing verification and Φ_{job} be the formula defined as the conjunction of all the formulae in Section 4, instantiated with respect to the given values of the parameters $\texttt{TOT_CORES}$, $\texttt{TOT_TASKS}_i$ for all the stages $i \in S$; and let $\texttt{DEADLINE}$ be a constant value defining the time bound for the job.

- *Feasibility analysis* aims to check whether **there exists** an execution of the system whose duration is lower than a given user-defined deadline. The CLTLoc formula corresponding to the property requires that all the stages completed their execution and the total elapsed time *totaltime* from the origin is less than the given deadline. The property φ_f is:

$$totaltime = 0 \land \mathbf{F} \left(\bigwedge_{i \in S} \texttt{completedS}_j \land (totaltime < \texttt{DEADLINE}) \right).$$

  The formula that is verified is $\Phi_{job} \land \varphi_f$. If the outcome of the satisfiability analysis is SAT then there exists an execution (*witness*) of the job that terminates earlier than $\texttt{DEADLINE}$ time units from the start. Conversely, if the result is UNSAT, there are no executions shorter than $\texttt{DEADLINE}$ or, equivalently, all the executions are longer than $\texttt{DEADLINE}$. Feasibility analysis does not require assumptions on the idle time of the CPU cores that is constrained by Formula 22, that can be omitted from the model.

- *boundedness analysis* aims to check whether the duration of **all** the executions of the system are shorter than a certain threshold. The CLTLoc formula corresponding to the property requires that if all the stages completed their execution then the total elapsed time *totaltime* from the origin is lower than the given deadline. The property φ_b is:

$$totaltime = 0 \Rightarrow \mathbf{G} \left(\bigwedge_{i \in S} \texttt{completedS}_j \Rightarrow (totaltime < \texttt{DEADLINE}) \right).$$

Copyright © 2017, DICE consortium – All rights reserved

The formula that is verified is $\Phi_{job} \wedge \neg\varphi_b$. If the outcome of the satisfiability analysis is SAT then there exists an execution (*counterexample*) of the job that terminates after `DEADLINE` time units from the start. Conversely, if the result is UNSAT, formula φ_b is a property of the modeled job; i.e., there are no executions longer than `DEADLINE` or, equivalently, all the executions are shorter than `DEADLINE`.

It is worthy to remark that boundedness analysis requires stronger assumptions on the idle time of the computational resources. Formula 22 limits the duration of the idleness of the CPU cores and, therefore, cannot be omitted from the model.

Copyright © 2017, DICE consortium – All rights reserved

6 Support for Spark in the Verification Tool

This section describes how the verification of Spark applications has been enabled in **D-VerT** by extending the **Json2MC** component in the **D-VerT** server. Given the modular nature of **Json2MC**, this extension involves the addition of a dedicated Spark module, as depicted in Figure 2. The module implements all the technology-specific aspects maintaining the same interface and is seamlessly integrated with the main component. In the current version, however, the support for Spark applications is still under development; in particular, **DTSM2Json** does not implement yet the model-to-model transformation of UML models representing Spark jobs.

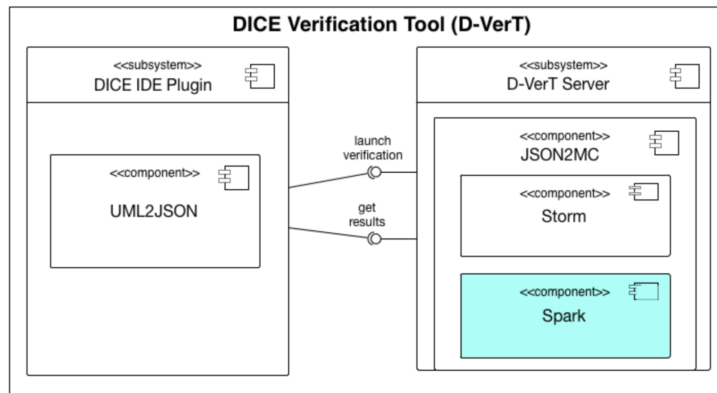


Figure 2: Simplified Component Diagram showing the main modules of **D-VerT**.

As shown in Figure 3, the module allows for the verification of temporal properties over Spark applications following the same workflow defined for Storm applications in deliverable D3.5 [3]. The user specifies first the UML design of the application from the Papyrus perspective of the DICE IDE. Next, he/she configures and launches the verification of the DIA directly from the DICE IDE by means of the **D-VerT** plugin. The **DTSM2Json** module, depending on the reference technology specified, extracts all the relevant features from the UML diagram and produces the corresponding JSON object containing all the information needed to run the verification. The JSON object is then used to invoke the **D-VerT** server, which, by means of the **Json2MC** module, will generate the technology-specific temporal logic model. The temporal logic model, consisting in a Lisp script, is then fed to **Zot**, the core bounded satisfiability/model checking tool that carries out the verification task. The outcome of the verification is processed back by **Json2MC**, which creates a graphical output and stores all the relevant input and output data. All of these statistics are then accessible by the client via the REST APIs that are exposed by the server.

Since the transformation from a UML model of the application is under development, currently the input for the tool consists of a Spark execution DAG, which can be obtained by profiling the application or be inferred from the code. In the next section, we will provide a simple use case to show the verification workflow on a toy application.

Copyright © 2017, DICE consortium – All rights reserved

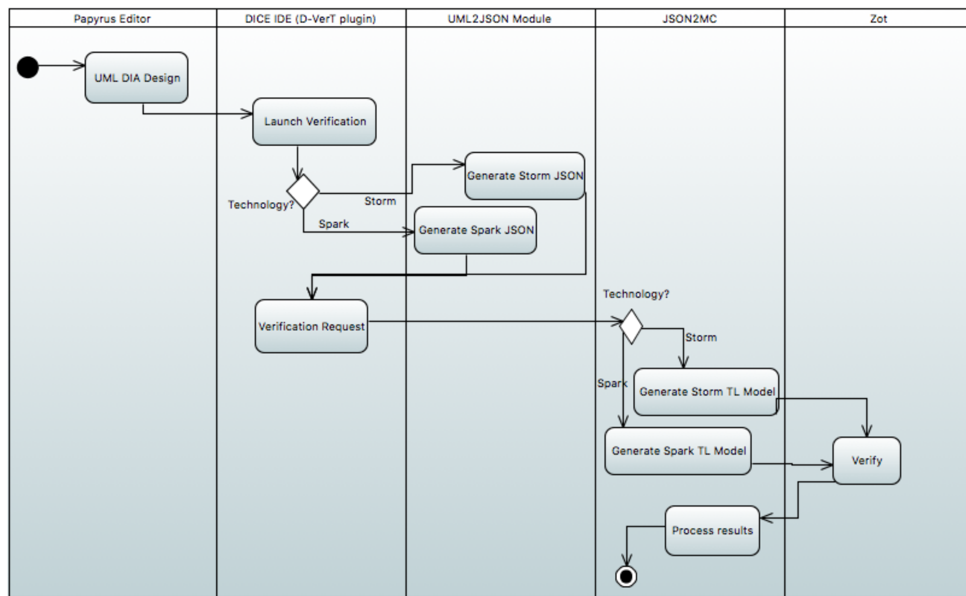


Figure 3: Activity Diagram representing the workflow from the UML design of the DIA to the execution of the verification task.

Copyright © 2017, DICE consortium – All rights reserved

7 Validation

This section presents the results of the validation activity of the **D-VerT** tool that was performed through a simple use case.

The Spark application in Listing 1 performs a set of simple operations over a text file where all the lines have the format: `type:name`. The final output is the list of all the `name` words having type different from `'verb'` and having the first letter equal to last letter.

Listing 1: Python code representing the toy 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)
```

When the Spark engine runs the code, it first generates the DAG entailed by the application code and then starts the processes to execute the stages. Figure 4 depicts the DAG that is associated with the code of Listing 1, in which each node corresponds to a stage. In general, any Spark application entails a DAG that can be obtained by analyzing the code (specifically, the dependencies among different RDDs and the transformations/actions performed over each RDD) without the need of running Spark.

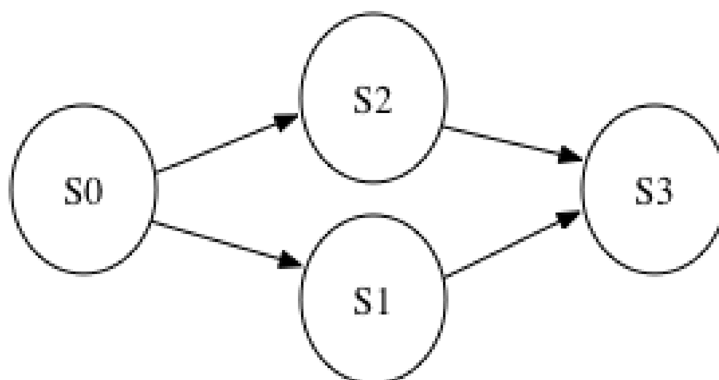


Figure 4: Execution DAG generated to run the application.

Copyright © 2017, DICE consortium – All rights reserved

An automatic procedure generating the DAG from the application code is currently under development. The DAG in Figure 4 has been generated by post-processing the log of Spark, but it can also be derived by an intuitive analysis of code in Listing 1. Stage `S0` includes the first set of transformations performed to define RDD `x`, that are concluded by the shuffle operation `reduceByKey()`. The latter implies data synchronization among all the partitions, therefore the computation cannot proceed without creating new stages. The results of `reduceByKey()` are then used to calculate the RDDs `y` and `z` with independent computations. These operations entail two more stages (`S1` and `S2`), both depending on the result of `S0`. These stages are symmetric and consist in applying `filter()`, `flatMap()`, `map()` and, finally, the shuffle operation `aggregateByKey()`. Finally, stage `S3` contain the set of operations to calculate `result`: `cartesian()`, `map()`, `filter()` and, finally, `collect()`. Being `collect()` an action, it determines the completion of the whole job.

As discussed in Section 4.2, each stage consists of a set of operations that can be executed in parallel on all the partitions of the input RDD and is performed by means of a number of tasks (each task applies the set of operations over a partition). We remark that the number of tasks that can be run in parallel depends on the number of cores available in the underlying homogeneous cluster. A stage can start only when all its predecessor stages are completed, and a stage can be defined as completed only when all its tasks terminated.

Listing 2 provides a simplified version of the JSON file—corresponding to the application DAG previously described—that is given as input to **Json2MC** in order to run the verification.

Listing 2: Simplified JSON file describing the execution DAG for the example application.

```
1  {
2  "app_name": "simple_app_example",
3  "verification_params": {
4      "plugin": "ae2sbvzot",
5      "time_bound": 30
6  },
7  "tot_cores": 600,
8  "analysis_type": "feasibility",
9  "deadline": 8.0,
10 "stages": {
11     "S0": {
12         "duration": 0.58,
13         "name": "reduceByKey at /test.py:6",
14         "numtask": 1000,
15         "parentsIds": []
16     },
17     "S1": {
18         "duration": 2.16,
19         "name": "aggregateByKey at /test.py:8",
20         "numtask": 500,
21         "parentsIds": ["S0"]
22     },
23     "S2": {
24         "duration": 1.9,
25         "name": "aggregateByKey at /test.py:10",
26         "numtask": 500,
27         "parentsIds": ["S0"]
28     },
29     "S3": {
30         "duration": 5.14,
31         "name": "collect at /test.py:12",
32         "nominalrate": 7.782101167315175,
33         "numtask": 500,
34         "parentsIds": ["S1","S2"]
35     }}}
36 }
```

Copyright © 2017, DICE consortium – All rights reserved

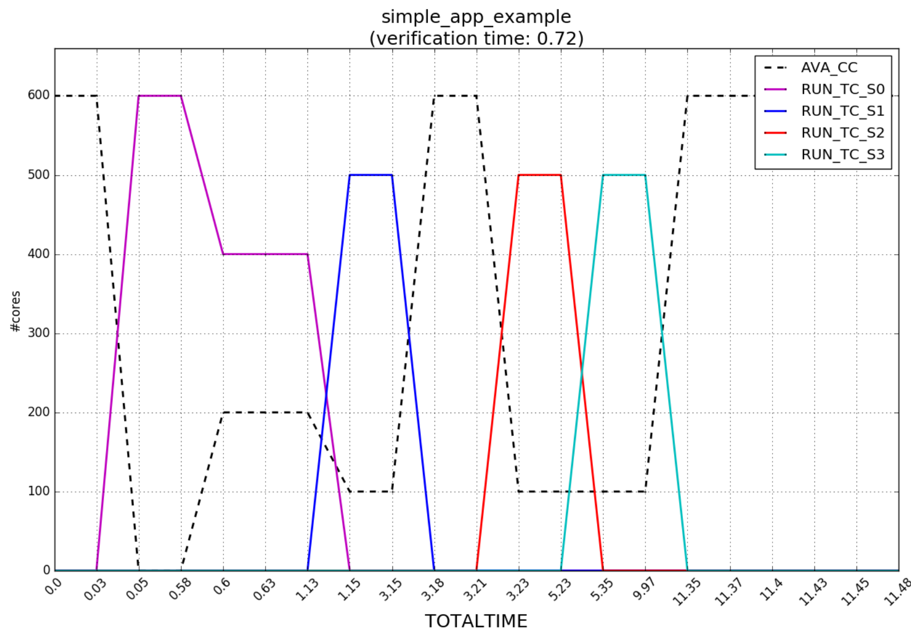


Figure 5: Graphical representation of the output trace produced by **D-VerT**.

Each stage is characterized by a number of tasks to be executed, the list of the parent stages (whose completion is needed to start the computation of the given stage), and the duration of each task. The other parameters contained in the JSON file are needed to configure the verification tool and set the verification problem. In the case of Listing 2, the JSON instructs **Json2MC** to verify if, having 600 cores available, there exists an execution of the application (`feasibility` analysis) which takes less than 8 milliseconds (`deadline` parameter) to complete. The check is performed by the *ae2sbvzot* plugin, whose verification approach is based on bounded satisfiability checking. Therefore, the tool requires a positive constant that specifies the number of the possible events that can occur in the executions modeled by the logical formulae. This value is often defined as a trade-off between the time needed to solve the satisfiability of the model and the number of events that one wants to consider in the executions. In this experiment, the bound was set to 30 as it allowed us to obtain meaningful results in a reasonable amount of time. The instantiated logical model turns out to be unsatisfiable (the tool outcome is UNSAT) when the deadline is 8ms, meaning that the job is not feasible as there does not exist an execution of the system taking less than 8ms (the specified deadline). However, if the deadline is set to 10 ms, the problem is satisfiable, and **Zot** produces an output trace describing a possible execution of the system whose duration is lower than the deadline. Since the output trace returned by **Zot** is in the text format, **D-VerT** produces a more user-friendly graphical representation that offers a visual hint of the execution. Figure 5 shows the output trace returned for the feasibility analysis of the application with the deadline set to 10.0 ms. The X-axis reports the timestamps of the different events occurring in the system, while the Y-axis represents either the number of cores that are used by each stage (colored solid line) or the number of available cores (black dotted line). Colored solid lines represent the execution status of the different stages, while the black dotted line plots the number of available cores at each time instant. It can be noticed that the last stage (S3) finishes its execution in the time instant 9.97, right below 10 milliseconds.

Copyright © 2017, DICE consortium – All rights reserved

8 Modeling and Verification of Storm Applications through First-Order Logic

This section describes a research activity whose preliminary results were presented in Deliverable D3.5, and which focuses on the application to DIAs of novel formal verification techniques able to handle systems with arbitrary numbers of objects. The research included two main activities. The first one was related to the practical aspects of verification, hence, to the study and examination of the tools for solving safety verification of systems defined with FOL and arrays. The second and more demanding one had the purpose of defining the FOL model capturing Storm applications for formal verification purposes. This second activity required (i) a study of the formalisms involved and of the verification algorithms implemented by the corresponding tools and (ii) a careful design of the model, which was achieved through an extensive testing campaign.

More precisely, the FOL model, which describes nodes running an unbounded number of threads, was used to carry out experiments with the MCMT[**mcmt**] and Cubicle[**cubicle**] tools, to evaluate the applicability of the approach. The trial-and-error experimental campaign showed the latter tool to be the best choice in terms of usability, but it required some bug fixing and extensions that needed various exchanges with the developers. The work improved the knowledge on the modeling capabilities of FOL specifications extended with arrays. It also helped to understand that the lack of a proper abstraction of Storm applications impedes the identification of a meaningful verification problem that can be defined in terms of safety verification of a FOL specification. In particular, the final FOL model was obtained after many steps of refinement that were needed to understand the resolution algorithm implemented by the tools and also to refine the way of modeling, with FOL and arrays, a given behavioral abstraction of DIAs.

In D3.5, the formalization and verification of DIAs based on the Apache Storm technology was proposed. Applications based on such technology can be abstracted by means of *topologies*, i.e., graphs representing the particularities of the application. The application topology, specified at DTSM level during the design phase, is transformed into the formal model to be verified through model checking techniques. Two classes of nodes define a topology. Input nodes (called *spouts*) are sources of information. Computational nodes (called *bolts*) process input data and produce results which, in turn, are emitted towards other bolts of the topology. Nodes in the topology can have also nonfunctional properties such as the size of data they produce, the emitting rate of data, etc. A topology also defines the connections among the nodes which allow the communication based on message exchange. Hence, any node is statically defined at design time by both the list of nodes subscribed to it and the list to which it subscribes.

The following paragraphs briefly recap the rationale for the abstraction modeling of the runtime behavior of Storm topologies.

The behavior of both spouts and bolts can be illustrated by finite state automata (see Figure 6). For more details on the terminology and modeling assumptions, the reader is directed to D3.5 (Section 4). A spout can be in one of the following states: $(I)dle$ (no tuples[3] are emitted.) or $(E)mit$ (the spout emits tuples to the bolts subscribed to it). A bolt can be in one of the following states: $(I)dle$ (no tuples are currently processed in the bolt), $E(X)ecute$ (at least one, and at most $Takemax$ tuples are processed at a certain time instance in the bolt), $ta(K)e$ (the bolt takes at least one, and at most $Takemax$, tuples from the queue and initializes a suitable number of concurrent threads to process them all), $(E)mit$ (the bolt emits tuples towards all the bolts that are subscribed).

Two modeling approaches have been proposed previously: a temporal logic model and a first-order logic (FOL) model. Here, the focus is on the second one. The FOL model was introduced as a complement to the temporal model given its ability to deal with an *arbitrary number of processes*, a feature that can be used to capture the parallelism in DIA components such as Apache Storm spouts and bolts. In the FOL model, DIAs are modeled as *array-based systems*. The specification of an array-based system composed of one array variable a and one transition τ consists of:

[3] Atomic data emitted or received by spouts and bolts

Copyright © 2017, DICE consortium – All rights reserved

(a) Finite automaton describing the states of a bolt (b) Finite automaton describing the states of a bolt

Figure 6: Finite automata describing the states of a spout/bolt

- a formula $Init(a)$ describing the initial sets of states, and
- a transition formula $\tau(a, a')$ relating a with an updated (modified) array variable a'.

A safety or reachability problem for the array-based system $S = (a, Init, \tau)$ is a formula $U(a)$ specifying the set of states the system should not be able to reach starting from a state in $Init$ and firing τ finitely many times. Therefore, in order to check the behavior of an array-based system, the set of initial states of the system and the action ordering in the system by a set of transitions are characterized. Both the initial state and the transitions introduce timing constraints for the time spent in each state.

In D3.5, as an example, the model of a simple Storm application composed of n replicas of a topology consisting of one spout and one bolt was proposed. The model should ensure that, for all processes, the length of the queue associated with a bolt does not exceed the maximum length $Lenmax$. For experiments, state-of-the-art model checkers MCMT[4] and Cubicle[5] were used. The outcome was that both can be used for our purposes, however *Cubicle allows for the definition of matrices, which can be a suitable abstraction*[6] *for modeling systems with m spouts and n bolts.*

Modeling assumptions. The models of the Storm applications presented in this section are compliant with the terminology and the assumptions introduced in D3.5, Sections 4.1.1 and Section 4.2, respectively, however they do not capture the failures of spouts and bolts.

Topology model. In the description of the topology model, we will introduce all the ingredients that are needed for the specification and verification of Storm applications using model checkers for infinite state systems as first introduced in [4].

The state of the spout with index i and process number x is indicated by the array variable $Spout(i, x)$. A spout can be in one of the two states: $(E)mit$ or $(I)dle$. Similarly, the state of the bolt with index i and process number x is indicated by a variable $Bolt(i, x)$. A bolt can be in one of the four states: $(I)dle$, $(E)mit$, $Ta(K)e$, $E(X)ecute$. Some other variables are maintained:

- $P(j, x)$ - number of tuples that are currently processed by process x in bolt j;
- $L(j, x)$ - the length of the bolt j in the process x;
- $s_{time}(j, x)$ - time units the spout j emits in the process x; a spout emits at least T^{spout}_{min} time units and at most T^{spout}_{max}. After the deadline T^{spout}_{max} another process can operate on the spout.
- $bEmitTakeTime(j, x)$ - time elapse between $P(j, x) = 0$ (which can happen in the states $Emit$ or $Execute$) and an $Emit$;
- $Taken(i, x)$ - the number of tuples taken by process x of bolt i;

Predicates $SubscribedBS(j, i)$ and $SubscribedBB(j, i)$ are used to describe the topology of the application.

- $SubscribedBS(j, i)$ - bolt with index j is subscribed to the spout with index i

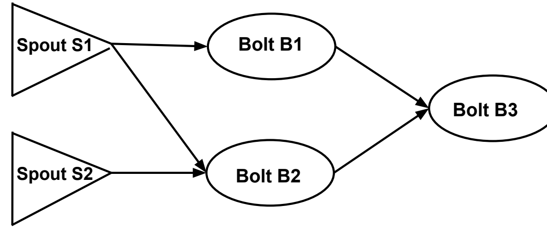
[4]http://users.mat.unimi.it/users/ghilardi/mcmt/

[5]http://cubicle.lri.fr/

[6]The model is called *counter networks*, was introduced in D3.5 and further developed in [**Marconi˙Storm˙2016**]

Copyright © 2017, DICE consortium – All rights reserved

Figure 7: Topology Example



- $SubscribedBB(j, i)$ - bolt with index j is subscribed to the bolt with index i.

Finally, the model includes a boolean variable $statechange$. If $statechange$ is true then either a spout or a bolt has to perform a state change; otherwise, when $statechange$ is false, components cannot vary the state and only time elapsing is allowed. This flag is used to enforce strict alternation between state changes and time elapsing so that topologies executions are realized by means of pairs of actions.

As an example, a topology consisting of 2 spouts and 3 bolts (like in the temporal logic model from D3.5) is considered (see Figure 7):

- B_1 is subscribed to S_1
- B_2 is subscribed to S_1 and S_2
- B_3 is subscribed to B_1 and B_2

In matrix form, that is:

$SubscribedBS$

	B_1	B_2	B_3
S_1	1	1	0
S_2	0	1	0

$SubscribedBB$

	B_1	B_2	B_3
B_1	0	0	1
B_2	0	1	0
B_3	0	0	0

The topology configuration above can be expressed as a set of invariants.

$$\forall_{i,j} \; i = 0 \wedge j = 0 \;\Rightarrow\; SubscribedBS[i, j] = True$$
$$\forall_{i,j} \; i = 1 \wedge j \geq 2 \;\Rightarrow\; SubscribedBB[i, j] = False$$
$$\ldots$$

The first formula states that the first bolt $B(0, x)$ is subscribed to the first spout $S(0, y)$. The second formula precisely states that the first bolt $B(0, x)$ is not connected to any other spout with index greater than or equal to 2.
The $Init$ state of the system is described by the formula:

$$t = 0.0 \wedge$$
$$\forall_{i,j} (S(i, j) = I \wedge B(i, j) = I \wedge L(i, j) = 0 \wedge P(i, j) = 0 \wedge s_{time}(i, j) = 0 \wedge bEmitTakeTime(i, j) = 0)$$

meaning that initially the clock is set to 0, all spouts and bolts are in the $(I)dle$ state, the length of all queues associated to the bolts and the number of tuples processed are 0, the value of s_{time} for all spouts is 0, time elapsed since $P(j, x) = 0$ and an $(E)mit$ of $P(j, x) = 0$ is 0.

In our experiments, the formal description of the topology composed of 2 spouts and 3 bolts is considered. Each transition is described by a logical formula that corresponds to the guarded assignment

Copyright © 2017, DICE consortium – All rights reserved

systems, correlating the values of state variables before and after the transition. We denote by X' the value of the variable X after the execution of the transition. For example, in the transition

$$\underset{x,y,i,j,c,d}{\exists} c > 0 \wedge B(i,x) = E \wedge L(j,y) + d \leq Lenmax(j) \wedge$$
$$\underset{l,z}{\forall} \begin{pmatrix} t' = & t + c & \wedge \\ L'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } L(l,z) + d \text{ else } L(l,z) & \wedge \end{pmatrix}$$

x, y, i, j are index variables, c, d are constants for elements of an array and of variables, respectively, B, L are tuples of array state variables, $B(i,x)$, $L(l,z)$, $Lenmax(j)$ (in B, L, respectively $Lenmax$) are the current values of a state variables and $B'(i,x)$, $L'(l,z)$ their values after the execution of the transition. Formula $c > 0 \wedge B(i,x) = E \wedge L(j,y) + d \leq Lenmax(j)$ is a conjunction of literals called *guard*, and the conjunctions of literals after $\underset{l,z}{\forall}$ is the *update* of the transition. t is a global variable, E is a value of the enumeration type $state$ representing the possible state of a spout/bolt.

Given the abstraction of the Storm application (topology in which a bolt node i has associated an information queue of maximum dimension $Lenmax(i)$), a suitable safety property to hold in the model is checking that states that satisfy the following formula are not reachable, where the formula is the negation of the property one wants to check and describes the sets of unsafe states:

$$\underset{x}{\exists} L(i,x) > Lenmax(i). \tag{23}$$

Note that (23) generalizes over the number i of processes (unbounded parallelism), distinct to the temporal logic model where the number of parallel processes active is specified.

Example 1. The first model[7] considered (Appendix A, Example 1 presents the transitions) consists of the following state variables: t, $statechange$, S, B, P, L, s_{time}, $bEmitTakeTime$ with the meaning give in Section 8. Additional, auxiliary, variables are $canTimeElapse$, $wasBTaking$. The set of 6 transitions[8], defining the model, is informally described bellow. Transitions $\sigma_{1a}, \sigma_{1b}, \sigma_4, \sigma_5$ allow $statechange$ according to Figure 6. For example, due to σ_{1a}, spouts $(E)mit$ or be $(I)dle$, bolts can be $(I)dle$ or perform $ta(K)e$ if previously were into $(E)mit$, or just in $ta(K)e$ if previously were $(I)dle$. In σ_{1b}, a process in a bolt can $(E)mit$ if, previously, it was processing tuples ($e(X)ecute$) and the number of tuples that are currently being processed by it is 0. Transition σ_4 and σ_5 capture the case when a bolt takes tuples. Two situations are possible: (1) if $Taken(j,y) \leq L(j,y)$, $L(j,y)$ decreases by the value of $Taken(j,y)$ and $P(j,y)$ becomes $Taken(j,y)$ (transition σ_4), (2) if $0 < L(j,y) < Taken(j,y)$ then $L(j,y)$ becomes 0 and $P(j,y)$ becomes $L(j,y)$ (transition σ_5).

Transition σ_2 captures the case when a spout emits. In the precondition of the transition, three conditions are checked: (1) if $SubscribedBS[j,i] = True$, case in which the length L of the queue associated to the bolt j is increased accordingly, (2) if the emitting time of the spouts is bounded by T^{spout}_{min} and T^{spout}_{max}, and (3) if the number of tuples added to the queue does not exceed $Lenmax(j)$.

Transition σ_3 captures the case when a bolt emits; if bolt $B(i,x)$ emits (1) there should be another bolt subscribed to it ($SubscribedBB(j,i) = 1$) and (2) the emitted information should not exceed the maximum length of the queue ($L(j,y) + d \leq Lenmax(j)$) case in which $L(j,y)$ is updated accordingly.

Transition σ_6 captures the behavior of the topology when the time simply passes: time is increased with c time units, the emitting time of a spout is increased with c and the number of tuples that are currently processed (state variable P) is decreased.

The verification of the safety property took about 1 second and the result was $Safe$ meaning that the system satisfies the property. However, this model is simplistic: in transition 3 (see Appendix A), tuples are added to the queue as long as we are below $Lenmax$, hence it can not happen that the safety property does not hold.

We refined the model above in many aspects because (1) the models were too weak or (2) the property was trivially either *true* or *false* or (3) the state space exploration that the backward reachability algorithm

[7] In all our experiments, we used Cubicle both for the formalization and verification. The Cubicle file for this examples is at `https://github.com/merascu/DICE-StormModellingFOL/blob/master/Example1.cub`

[8] The Cubicle file contains more transitions since we had to tune the model to system capabilities. For example capturing that $S'(k,z) = E$ or I led to two distinct transitions.

Copyright © 2017, DICE consortium – All rights reserved

implemented in the tool Cubicle has to perform is infeasible. By trial-and-error we found a model (see Example 2 below) which is useful to prove the safety property above.

Example 2. After many refinements of the initial model, a model[9] (see Appendix A, Example 2) in which the safety property (23) holds is obtained only for one bolt, that is $B(1, x)$, is obtained. For simplifications, we made the following assumptions:

1. $Lenmax(i)$ = 3.5, $Takemax[i]$ > 3, $Tsmin$ = 9.2, $Tsmax$ = 9.8;
2. one process processes one tuple
3. the number of tuples taken represent the number of active threads/processes in a bolt at a certain time.
4. queues have only one dimension, i.e., there is one queue for each bolt meaning that processes in a bolt share the same queue;
5. different possible states for the spouts are left out; one keeps track of the time elapsing to enable spout emit (s_{time});
6. $(E)mit$ of a bolt is enforced if a bolt is ready to emit;
7. statechange from $(I)dle$ to $ta(K)e$ is enforced if a bolt has non-empty queue and some processes are $(I)dle$.

In the model, there are 3 transitions defining the behavior of the topology: $spout_{emit}$, $bolt_{emit}$ and $bolt_{take}$; each of them can fire only if the associated flag, that is $DoEmit$ - for bolt/spout emit, $DoTake$ - for bolt take, is true. In order to set correctly the flags we used three transitions per flag $setDoTake_T$, $setDoTake_{F1}$ and $setDoTake_{F2}$, one transition defining $DoTake = True$ and two for $DoTake = False$. A careful reader might wonder why did we need such a complicated way of triggering actions. The answer lies in the backward reachability algorithm which is implemented in Cubicle and it is not devised for verifying systems where some transitions must occur. In timed automata verification, for instance, some tools implement this kind of must transition; namely, if a "must" transition is enabled then only that transition is fired first and all the other that are enabled are fired later. In our experiments we could not force the firing of a transition even if its guard was satisfied. This is the reason why we had to implement these "must" transitions, i.e. 6 "auxiliary" transitions setting the two flags correctly. Moreover, to keep the state space amenable, we limited the executions to a single bolt to make them compliant with this alternating mechanisms, which sets $DoTake$ and $DoEmit$.

An example of "auxiliary" transition setting $DoTake$ to $True$ is as follows.

$$\begin{aligned} setDoTake_T : \quad & \exists_{i,x}\ i = 1 \land 0 \le x \land x < Takemax[i] \land L[i] > 0 \land B[i,x] = I \quad \land \\ & SetFlags = DT \ \land \ DoTake = True \ \land SetFlags = DE \end{aligned}$$

The remaining 4 transitions, 3 defining the behavior of the topology and 1 time elapse.

- spout emit (transition $spout_{emit}$): the queue of the bolts subscribed to the spout increases and the emit time of the spout is reset,
- bolt emit (transition $bolt_{emit}$): changes the state of the bolts to $(I)dle$ and the bolts subscribed to it increase their queue,
- bolt take (transition $bolt_{take}$): the length of the corresponding queue is decreased,
- time elapse (transition $time_{elapse}$: if the bolt is in state $e(X)ecute$ it changes either to $(E)mit$ or $(I)dle$, the number P of processed tuples decreases and the emitting time of the spout increases (by c).

[9]Cubicle file at `https://github.com/merascu/DICE-StormModellingFOL/blob/master/Example2.cub`.

Copyright © 2017, DICE consortium – All rights reserved

We ran this model for around 12 hours obtaining $Safe$, meaning that the system satisfies the property. This version of the model captures some of the features of a Storm application, but it can only be applied to very simple applications. Some of the assumptions made above were imposed by the tools we used: restrictions of the backward reachability algorithm and also restrictions of the input language of the tools, which led to a significantly big number of transitions in the model.

By analyzing the result obtained from this modeling exercise, as well as the model per se, we conclude that it is worth verifying Storm applications with safety properties which generalize over the number of processes, since more processes lead to faster applications, but also to greater resource consumption. Formula (23) is a particular example of such a property. However, further investigation is needed in order to find a better abstraction for DIAs, one which does not impose too many restrictions stemming from very limiting assumptions; in summary, an abstraction that might be able to express more realistic applications.

Applying first-order logic techniques to the verification of DIAs is a challenge, in no small part due to the relative lack of maturity of the supporting tools; the goal in DICE is to investigate these techniques as a possible complement to the temporal logic-based mechanisms which are widely applied in practice. First-order logic-based techniques have been successfully applied in academic research to validate protocols with unbounded number of concurrent threads (e.g., Fischer protocol)[10] [11] and can also be considered as a candidate approach for static analysis, limited to safety aspects of array-manipulating code. Inspired by these positive results, we investigated the application of these techniques to the verification of safety properties of DIAs. The results we obtained highlight the current limitations of the techniques, which hamper their applicability in practice to realistic systems, and the need for suitable abstractions that can be useful for the analysis of DIAs with FOL. Moreover, our experiments open the possibility of creating synergies between researchers from automated reasoning and practitioners from the Big Data community.

[10] `http://users.mat.unimi.it/users/ghilardi/mcmt/home.html`
[11] `http://cubicle.lri.fr/#experiments`

Copyright © 2017, DICE consortium – All rights reserved

9 Conclusion and future works

In this section we provide a wrap-up of what has been accomplished so far with the development of the DICE verification framework.

The main achievements of this deliverable in relation to the initial requirements for the tool are shown in Table 1. Our activities have been focused first on the fulfillment of requirements **R3.1** and **R3.2** as they were necessary to integrate **D-VerT** with the DICE IDE. A detailed description of the definition and development of the model-to-model transformations can be found in deliverable **D3.1**[5]. Once the transformations needed to carry out the verification of Storm applications has have been defined, we focused on the integration with the DICE IDE by targeting all the IDE-related requirements. Moreover, we extended our approach to the Spark technology. Advancements related to **R3.12** consisted in the definition of a formal model to support the analysis and verification of temporal properties for Spark applications. As described in Sect. 4, the model captures the runtime behavior of the Spark framework and can be instantiated by specifying the execution DAG corresponding to an application. In the current stage of development, the verification can be performed by providing a JSON file containing the DAG specification and some tool and property configurations. Future work will address the integration of the Spark-related functionalities with the IDE and the possibility of expressing the application in a more user-friendly way (e.g. by means of UML models).

Requirement ID	Description	Coverage	To do
R3.1	M2M Transformation	60 %	Spark transformations in DICE IDE
R3.2	Taking into account relevant annotations	60 %	New annotations for Spark.
R3.3	Transformation Rules	0 %	
R3.7	Generation of traces from system model	80 %	Integration in the DICE IDE for Spark
R3.10	SLA specification and compliance	30 %	Highlighting violated SLA
R3.12	Modelling abstract level	90 %	Refinements to the temporal logic models
R3.15	Verification of temporal safety/privacy properties	70 %	Support for Spark in the IDE.
R3IDE.2	Timeout Specification	50 %	Integration in the DICE IDE
R3IDE.4.2	Loading the properties to be verified	60 %	Integration for Spark
R3IDE.5	Graphical output	80 %	Integration in the DICE IDE for Spark
R3IDE.5.1	Graphical output of erroneous behaviours	80 %	Integration in the DICE IDE for Spark

Table 1: Requirement coverage at month 24.

9.1 Further work

Starting from the requirements listed in Table 1, the following items provide an overview of the next issues to be addressed within Task T3.3 and of the forthcoming work that will be carried out until M30.

IDE : In order to meet the requirements related to the IDE, most of the effort will be addressed to the development of the functionalities enabling the analysis and verification of Spark applications

Copyright © 2017, DICE consortium – All rights reserved

(**R3IDE.4.2**, **R3IDE.5** and **R3IDE.5.1**). We plan to fulfill **R3IDE.2** by adding to the client the possibility to set a timeout for the verification tasks.

R3.1, R3.2, R3.7 : The same approach adopted for Storm applications will be exploited to develop the M2M transformations from UML diagrams describing Spark application (with a specific) to the JSON needed to perform verification.

R3.10 : Additional work is needed to support the definition of quality SLAs against which run the verification tasks.

R3.12 Further work will address the refinement and improvement of abstract models. On one hand, in order to decrease the verification time, we plan to review the existing models in order to reduce their size and possibly also their complexity. O the other hand we will try to add more details in order to carry out a finer-grained analysis of the systems.

R3.15 : beyond the efforts in extending the functionalities in the IDE to support Spark applications, furher investigations will be devoted to the possibility of modeling and verifying further properties.

Copyright © 2017, DICE consortium – All rights reserved

References

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

[2] 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.

[3] M. Bersani et al. *DICE Verification Tool - Initial Version*. Tech. rep. www.dice-h2020.eu. DICE Consortium, 2016.

[4] S. Ghilardi et al. "Towards SMT Model Checking of Array-Based Systems". In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. 2008, pp. 67–82.

[5] M. Bersani M. Erascu A. Gómez C. Joubert F. Marconi J. Merseguer J. I. Requeno D. Ardagna S. Bernardi and M. Rossi. *DICE Transformations to analysis models*. Tech. rep. www.dice-h2020.eu. DICE Consortium, 2016.

Copyright © 2017, DICE consortium – All rights reserved

A Details of the Formal Models

A.1 First Order Logic Model. Example 1

$$\sigma_{1a}: \exists_{x,y,i,j}\ statechange = 1 \wedge \forall_{l,k,z} \left(\begin{array}{lll} statechange' = & 0 & \wedge \\ S'(k,z) = & \text{if } (z = x \wedge k = i) \text{ then } (E \text{ or } I) \text{ else } S(k,z) & \wedge \\ B'(l,z) = & \text{if } (z = y \wedge l = j \wedge B(l,z) = E) \text{ then } (I \text{ or } K) \text{ else } B(l,z) & \\ & \text{elseif } (z = y \wedge l = j \wedge B(j,z) = I) \text{ then } K \text{ else } B(l,z) & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_{1b}: \exists_{j,y}\ B(j,y) = X \wedge P(j,y) = 0 \wedge \forall_{l,z} \left(\begin{array}{lll} statechange' = & 0 & \wedge \\ B'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } E \text{ else } B(l,z) & \wedge \\ wasBEmitting' = & 1 & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_2: \exists_{x,y,i,j,c,d}\ c > 0 \wedge d > 0 \wedge S(i,x) = E \wedge T^{spout}_{min} < s_{time}(i,x) + c < T^{spout}_{max} \wedge$$
$$L(j,y) + d \leq Lenmax(j) \wedge SubscribedBS(j,i) = 1 \wedge$$
$$\forall_{l,k,z,t} \left(\begin{array}{lll} t' = & t + c & \wedge \\ statechange' = & 1 & \wedge \\ L'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } L(l,z) + d \text{ else } L(l,z) & \wedge \\ P'(l,z) = & \text{if } B(l,z) = X \wedge P(l,z) - Execrate(l) * c \geq 0 & \\ & \text{then } P(l,z) - Execrate(l) * c \text{ else } 0 & \wedge \\ s'_{time}(k,t) = & \text{if } (t = x \wedge k = i) \text{ then } 0 \text{ else } s_{time}(k,t) + c & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_3: \exists_{x,y,i,j,c,d}\ c > 0 \wedge B(i,x) = E \wedge L(j,y) + d \leq Lenmax(j) \wedge SubscribedBB(j,i) = 1 \wedge$$
$$\forall_{l,z} \left(\begin{array}{lll} t' = & t + c & \wedge \\ statechange' = & 1 & \wedge \\ L'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } L(l,z) + d \text{ else } L(l,z) & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_4: \exists_{j,y}\ B(j,y) = K \wedge Taken(j,y) \leq L(j,y) \wedge$$
$$\forall_{l,z} \left(\begin{array}{lll} statechange' = & 0 & \wedge \\ B'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } X \text{ else } B(l,z) & \wedge \\ L'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } L(l,z) - Taken(j,y) \text{ else } L(l,z) & \wedge \\ P'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } Taken(j,y) \text{ else } P(l,z) & \wedge \\ bEmitTakeTime'(l,z) = & 0 & \wedge \\ wasBTaking' = & 1 & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_5: \exists_{j,y}\ B(j,y) = K \wedge 0 < L(j,y) < Taken(j,y) \wedge$$
$$\forall_{l,z} \left(\begin{array}{lll} statechange' = & 0 & \wedge \\ B'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } X \text{ else } B(l,z) & \wedge \\ L'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } 0 \text{ else } L(l,z) & \wedge \\ P'(l,z) = & \text{if } (z = y \wedge l = j) \text{ then } L(l,z) \text{ else } P(l,z) & \wedge \\ bEmitTakeTime'(l,z) = & 0 & \wedge \\ wasBTaking' = & 1 & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

Copyright © 2017, DICE consortium – All rights reserved

$$\sigma_6: \exists_c\, c > 0 \wedge canTimeElapse = 1 \wedge \forall_{j,z} \left(\begin{array}{lll} t' = & t + c & \wedge \\ statechange' = & 1 & \wedge \\ P'(j,z) = & \text{if } (B(j,z) = X \;\wedge\; P(j,z) - Execrate(j) * c \geq 0) & \\ & \text{then } P(j,z) - Execrate(j) * c & \\ & \text{elseif } B(j,z) \neq X \text{ then } P(j,z) \text{ else } 0 & \wedge \\ s'_{time}(j,z) = & s_{time}(j,z) + c & \wedge \\ bEmitTakeTime\prime(j,z) = & bEmitTakeTime(j,z) + c & \wedge \\ canTimeElapse' = & 0 & \end{array} \right)$$

A.2 First Order Logic Model. Example 2

We use two enumeration types: for representing states (I, E, X) and flags for signaling the states $ta(K)e$ (DT), $(E)mit$ (DE), or none of these two (No).

$$setDoTake_T: \quad \exists_{i,x}\, SetFlags = DT \wedge i = 1 \wedge 0 \leq x < Takemax(i) \wedge \;\; DoTake' = True \wedge SetFlags' = DE$$

$$setDoTake_{F1}: \quad SetFlags = DT \wedge j = 1 \wedge L(j) = 0 \wedge DoTake' \;\; DoTake' = False \wedge SetFlags' = DE$$

$$setDoTake_{F2}: \quad \exists_j\, SetFlags = DT \wedge j = 1 \wedge L[j] > 0 \wedge 0 \leq j < Takemax(j) \wedge B(j,j) \neq I) \wedge$$
$$\forall_y (y < 0 \vee y \geq Takemax(j) \vee B(j,y) \neq I) \wedge \;\; DoTake' = False \wedge SetFlags' = DE$$

$$setDoEmit_T: \quad \exists_{i,x}\, SetFlags = DE \wedge i = 1 \wedge 0 \leq x < Takemax(i) \wedge P(i,x) = 0 \wedge B(i,x) = E \wedge$$
$$DoEmit' = True \wedge SetFlags' = No$$

$$setDoEmit_{F1}: \quad \exists_j\, SetFlags = DE \wedge j = 1 \wedge 0 \leq j < Takemax(j) \wedge B(j,j) \neq E \wedge$$
$$\forall_y (y < 0 \vee y \geq Takemax(j) \vee B(j,y) \neq E) \wedge \;\; DoEmit' = False \wedge SetFlags' = No$$

$$setDoEmit_{F2} \quad \exists_j\, SetFlags = DE \wedge j = 1 \wedge 0 \leq j < Takemax(j) \wedge B(j,j) = E \wedge P(j,j) > 0 \wedge$$
$$\forall_y (y < 0 \vee y \geq Takemax(j) \vee (B(j,y) = E \wedge P(j,y) > 0.0)) \wedge$$
$$\wedge \;\; DoEmit' = False \wedge SetFlags' = No$$

Copyright © 2017, DICE consortium – All rights reserved

$$
\begin{aligned}
spout_{emit}: \quad & \exists_{i,j}\, Tsmin < s_{time}[i] \;\wedge\; SubscribedBS[j,i] = True \;\wedge\; DoTake = False \;\wedge \\
& DoEmit = False \;\wedge\; SetFlags = No \;\wedge \\
& \forall_{l} \begin{pmatrix}
L'[l] = & \text{if } (l = j) \text{ then } L[l] + 1 \text{ else } L[l] & \wedge \\
s'_{time}[l] & \text{if } (l = i) \text{ then } 0 \text{ else } s_{time}[l] & \wedge \\
CanTimeElapse' = & True & \wedge \\
SetFlags' = & DT &
\end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
bolt_{emit}: \quad & \exists_{i,j,x}\, B[i,x] = E \;\wedge\; SubscribedBB[j,i] = True \;\wedge\; 0 \le x < Takemax[i] \;\wedge\; DoEmit = True \;\wedge \\
& \forall_{l} \begin{pmatrix}
L'[l] = & \text{if } (l = j) \text{ then } L[l] + 1 \text{ else } L[l] & \wedge \\
B'[l,z] = & \text{if } (z = x \wedge l = i) \text{ then } I \text{ else } B[l,z] & \wedge \\
CanTimeElapse' = & True & \wedge \\
SetFlags' = & DT &
\end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
bolt_{take}: \quad & \exists_{j,y}\, B[j,y] = I \;\wedge\; L[j] \ge 1 \;\wedge\; 0 \le y < Takemax[i] \;\wedge\; DoTake = True \;\wedge \\
& \forall_{l} \begin{pmatrix}
L'[l] = & \text{if } (l = j) \text{ then } L[l] - 1 \text{ else } L[l] & \wedge \\
B'[l,z] = & \text{if } (z = y \wedge l = j) \text{ then } X \text{ else } B[l,z] & \wedge \\
P'[l,z] = & \text{if } (z = y \wedge l = j) \text{ then } 1 \text{ else } P[l,z] & \wedge \\
CanTimeElapse' = & True & \wedge \\
SetFlags' = & DT &
\end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
time_{elapse}: \quad & \exists_{j,y}\, 0 < c \;\wedge\; CanTimeElapse = True \;\wedge \\
& \forall_{l} \left(\begin{array}{ll}
P'[l,z] = & \text{if } (B[j,z] = X \wedge 0 \le P[j,z] - c) \text{ then } P[j,z] - c \text{ else } 0 \wedge \\
B'[l,z] = & \text{if } (B[j,z] = X \wedge j = 0 \wedge 0 \le z < Takemax[j] \wedge P[j,z] \le c) \text{ then } E \\
& \text{elseif } (B[j,z] = X \wedge j = 1 \wedge 0 \le z < Takemax[j] \wedge P[j,z] \le c) \text{ then } E \\
& \text{elseif } (B[j,z] = X \wedge j = 2 \wedge 0 \le z < Takemax[j] \wedge P[j,z] \le c) \text{ then } I \\
& \text{else } B[j,z] \wedge \\
s'_{time}[j] & s_{time}[j] + c \wedge \\
CanTimeElapse' = & False \wedge \\
SetFlags' = & DT
\end{array}\right.
\end{aligned}
$$

Copyright © 2017, DICE consortium – All rights reserved