Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE Verification Tools - Initial Version

Deliverable 3.5

Deliverable:	D3.5
Title:	Verification Tools - Initial Version
Editor(s):	Matteo Rossi.
Contributor(s):	Marcello M. Bersani, Madalina Erascu, Francesco Marconi
Reviewers:	Giuliano Casale, Simona Bernardi
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-Jan-2016
Status:	Final version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2016, DICE consortium – All rights reserved

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

Executive summary

This document is the first 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 user-defined properties, in particular safety ones, such as reachability of undesired configurations of the system, meeting of deadlines, and so on. More precisely, **D-VerT** takes a DICE annotated model and the property to be verified and converts them into a formal model (a temporal logic model or a First-Order Logic model). Based on the type of property to be verified (in the case of non-expert users) or on the verification approach the user applies (in case of expert users) the appropriate verification method is applied. The result is presented in the form of *Yes/No* answer and in case of *No*, the trace of the system that violates it.

D-VerT encompasses the **Verifier** component composed of two subcomponents **DTSM2Json** and **Json2MC** and utilizes state of the art model checkers, e.g. **Zot**[1], **MCMT**[2] and **Cubicle**[3], for the actual formal verification.

In this document, we present the status of the development activities related to task T3.3 and some instructions install and use the prototype tool that is being developed in the framework of this task. At this stage, the **Json2MC** component has been defined, which has the role of translating a DICE Platform and Technology Specific Model (DTSM), captured through a simple the JSON format, into a formal model. In addition, we also defined two new models for Storm applications, which have been validated using **D-VerT**. This activity required either extending the features of the model checkers, or coming up with novel ideas how to address scalability.

The current prototype represents a core component of the DICE solution. This document also includes a coverage assessment of the requirements related to the Verification Tools (which have been defined in deliverable D1.2).

[1]`https://github.com/fm-polimi/zot`
[2]`http://users.mat.unimi.it/users/ghilardi/mcmt/`
[3]`http://cubicle.lri.fr/`

Glossary

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

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The focus of the DICE project is to define a quality-driven framework for developing data-intensive applications (DIAs) that leverage Big Data technologies hosted in private or public clouds. DICE offers a novel profile and tools for data-aware quality-driven development. DICE-profiled models are fed into a set of simulation, verification and optimization tools to obtain high-quality applications. One of these tools within the DICE framework is the so-called DICE Verification Tool (**D-VerT**), which allows application designers to evaluate their design against user-defined properties, in particular safety ones, such as reachability of undesired configurations of the system, meeting of deadlines, and so on. This document describes the initial version of **D-VerT**, which is developed in the framework of WP3 Task 3.3 and which is published as an open source tool in the DICE-Verification repository of the project github[4].

1.1 Objectives

The *goal of WP3* is to develop a quality analysis tool-chain that will be used to guide the early design stages of DIAs and the evolution of the quality of the application once operational data becomes available. The main outputs of these tasks are tools *(i)* for simulation-based reliability and efficiency assessment, *(ii)* for formal verification of safety properties related to the sequence of events and states that the application undergoes, and *(iii)* numerical optimisation techniques for the search of optimal architecture designs. WP3 defines model-to-model (M2M) transformations that accept as input the design models defined in tasks T2.1 and T2.2 and produce as outputs the analysis models used by the quality tools.

Task T3.3 focuses on the development of a verification framework to evaluate safety properties of DIAs specifically related to so-called Big Data topologies. The task also considers the verification of privacy properties with a low amount of effort.

The outcome of tasks T2.1 and T2.2 are DIA topologies, which consist of two parts, defined in the DTSM annotated diagrams: *(i)* the graph-based representation of the topology under analysis and *(ii)* a set of user-defined non-functional parameters which complement the graph-based representation and allow the designer to perform the verification.

This deliverable reports on the activity carried out in Task T3.3, whose goal is to develop and implement a set of tools (which could be extensions of existing ones) supporting verification of DIA topologies. Real-time temporal properties are specified through suitable extensions of Linear Time Temporal Logic and fragments of First-Order Logic (FOL) with arrays. A fundamental goal of T3.3 is the implementation of a verification environment allowing the DIA designer to easily select the properties to verify from a list of –possibly predefined– properties concerning the DIA topology. The document provides an architectural and behavioural description of the tool, which is the reference schema for D1.3 and D1.4 (i.e., architecture definition and integration plan, initial version and final version, respectively). It shows the model-to-model transformations required to translate a description of a DIA topology into a model which undergoes verification, a first relevant class of properties to verify, and an initial demonstrator of the tools being developed in Task T3.3, which implements this translation. The demonstrator provides an initial working prototype of **D-VerT**.

The contribution of Task T3.3 consists of the definition of a formal model of DIAs, and of a new class of non-deterministic models to represent DIAs, which are studied from the point of view of their theoretical aspects and which are compared with other similar formalisms, e.g., Timed Petri nets [1] and Queueing networks [2]. The peculiarity of the class of models defined in Task T3.3 is their inherent non-determinism, suitably captured through formalisms such as temporal logic. This complements the modelling adopted in Task T3.1, which is based on stochastic formalisms, suitable to obtain performance and optimization analysis and for simulation purposes. The main advantage of using temporal logic specifications instead of Timed Petri Nets lies in their expressiveness.

- Timed Petri Nets cannot model the following behaviour of a system: *an event may (but has not to) occur in one time unit from the current instant*. This result intuitively stems from the *urgent*

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

semantics adopted in almost all the tools and theoretical works for Timed Petri Net [3]. Given an upper and lower timing bounds $a \leq b$, *urgent* semantics enforces the firing of the transition labelled with $[a, b]$ no later than b, but at least after a time units after the transition is enabled. The modelling approach proposed in Section 4, instead, specifies that a failure *may* (but need not) occur only after a certain time delay following the previous one.

- The standard semantics of the firings of Timed Petri Nets does not allow the modelling of the following queue policy: *dequeuing always removes the maximum number of available elements in the queue, but never more than k elements at the same time*. The model of Section 4 makes use of this abstraction to represent the behaviour of a node when it extracts new elements from its queue to process them.

1.2 Motivation

Safety verification of topologies is performed to check the reachability of bad configurations, i.e., a malfunction of the application which consists of behaviours that do not conform to some non-functional requirements specified by the QA_ENGINEER. Malfunctions can be generated by various factors in the deployed application. Task T3.3 focuses on the analysis of the effect of node failures and incorrect design of timing constraints which might cause the following anomalies: *(i)* latency in processing tuples, and *(ii)* monotonic growth of queues occupancy.

Verification does not consider the cause of a node failure but its effect on the requirements of the application. After a node failure, the total delay that the topology requires to process one tuple (or a set of tuples) can exceed the maximum tolerated delay, specified in the requirements of the application, or the quality of the information processing may degrade. Underestimating the computational power of nodes also affects the time to process tuples, which then might not meet the timing requirements of the application, as the processing delay might cause the saturation of queues of the message system. The analysis does not take into account the quality aspect of the processing, and it focuses instead on the temporal aspects of the implemented topology.

DICE adopts model checking techniques to verify topologies specified in DTSM diagrams. A verification problem is specified by a formal description of the DIA topology and a logical formula representing the property that its executions must satisfy. The DICE verification activities rely on two approaches: a *fully automatic one* based on dense-time temporal logic with discrete variables and realized according to the bounded model-checking approach; and a *semi-automatic one* which is a state-based approach relying on FOL over arrays and backward reachability analysis of unsafe states. The verification process is designed to be carried out in an agile way; it should allow the DIA designer to perform verification tasks using a lightweight approach. More precisely, **D-VerT** fosters an approach whereby formal verification on DIA applications is launched through interfaces that hide the complexity of the underlying models and engines. These interfaces allow the user to easily produce the formal model to be verified and the properties to be checked. This eliminates the need for the user to be an expert of the formal verification techniques on which **D-VerT** is founded. In addition, the outcome of the verification process is shown back to the user in a graphical manner through –possibly UML-compliant– diagrams, as described in the following sections. Semi-automatic verification, on the other hand, is tailored for advanced users.

1.3 Structure of the document

The structure of this deliverable is as follows. Section 2 recaps the requirements related to the Verification Tools. Section 3 discusses the architecture of **D-VerT** and shows its interaction with the DICE framework. Section 4 provides some context about DIAs, the assumptions and the design of the formal model and its definition in logic. Section 5 discusses the implementation of **D-VerT**, with particular focus on the **Json2MC** component. Section 6 shows some practical experiments to validate the approach and discusses future achievements. Finally, Appendix A provides some additional details on the formal models.

2 Requirements and usage scenarios

Deliverable D1.2 [4, 5] 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 as input 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 [4, 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 [4] are the following:

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

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

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

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

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

ID:	R3.12
Title:	Modelling abstraction level
Priority of accomplishment:	Must have
Description:	Depending on the abstraction level of the UML models (detail of the information gathered, e.g., about components, algorithms or any kind of elements of the system we are reasoning about), the TRANSFORMATION_TOOLS will create the formal model accordingly, i.e., at that same level that the original UML model

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

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

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

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

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

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

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

3 Verification tool overview

D-VerT (DICE Verification Tool) is the verification tool integrated in the DICE framework. Verification is performed on annotated DTSM models which contain all the information required to perform the analysis. The user selects a –safety/privacy– property to be checked, possibly using templates (which are compliant with the class of properties specified in the design phase at DPIM level).

The DTSM annotated model and the property to be verified are converted into a formal model that is suitable for verification. Based on the type of the property to verify (i.e., safety or privacy) and on the type of model the user specifies (i.e., temporal logic model or FOL model), the tool selects the appropriate solver. The outcome is sent back to Verifier-GUI and then to the IDE, which presents the result. It shows whether the property is fulfilled or not; and, if it is violated, the IDE presents the trace of the system that violates it.

The sequence diagram related to the verification phase which shows the interaction of **D-VerT** with two DICE components, IDE and Verifier-GUI, is depicted in Fig. 1.

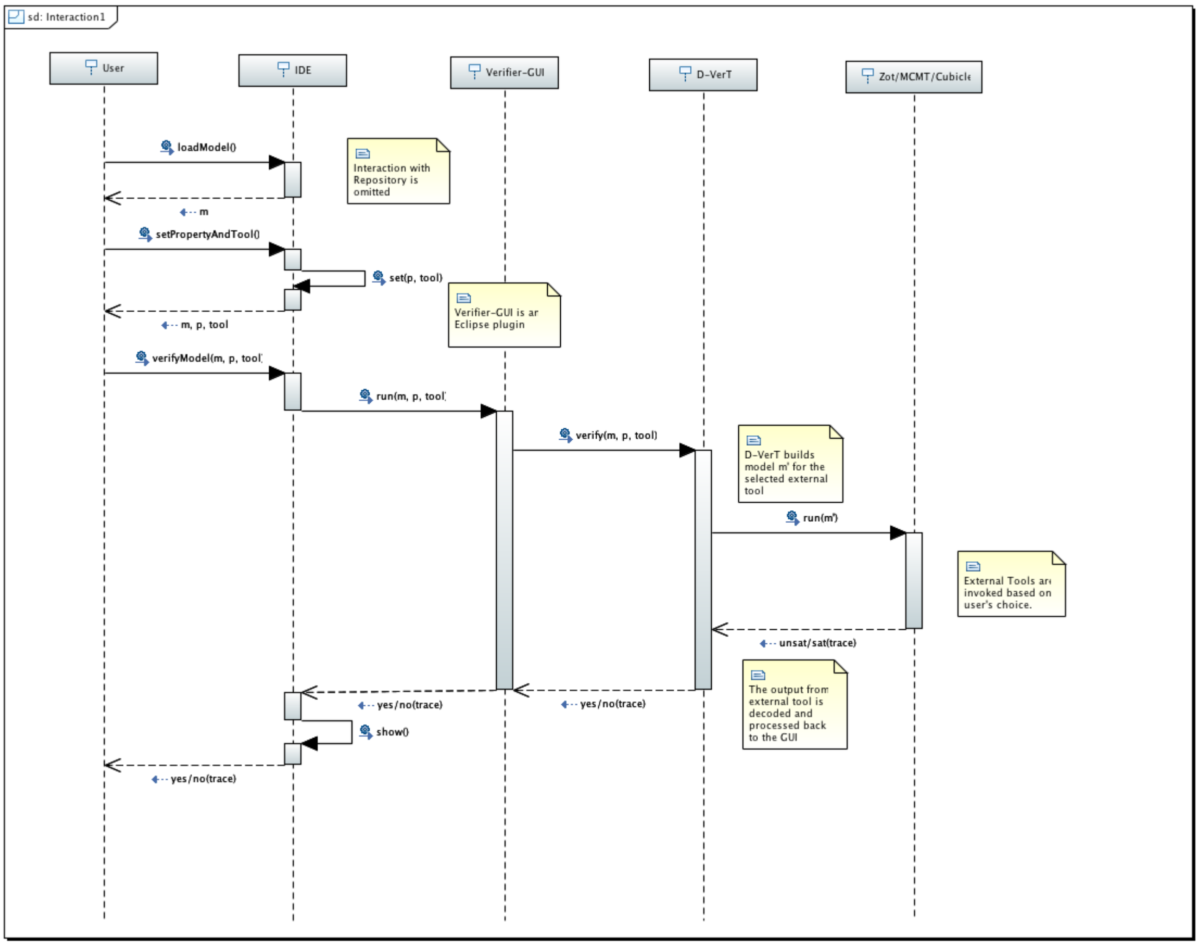

sd: Interaction1
User
IDE
Verifier-GUI
D-VerT
Zot/MCMT/Cubicle
loadModel()
Interaction with Repository is omitted
m
setPropertyAndTool()
set(p, tool)
Verifier-GUI is an Eclipse plugin
m, p, tool
verifyModel(m, p, tool)
run(m, p, tool)
verify(m, p, tool)
D-VerT builds model m' for the selected external tool
run(m')
External Tools are invoked based on user's choice.
unsat/sat(trace)
The output from external tool is decoded and processed back to the GUI
yes/no(trace)
yes/no(trace)
show()
yes/no(trace)

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

D-VerT is the core component that generates the model to verify. The latter is defined according to the rationale explained in Section 4. The input of **D-VerT** is a DTSM annotated diagram and the tool produces a script file which undergoes verification by means of external model-checkers. **D-VerT** is constituted of two subcomponents as shown in Fig. 2.

- **DTSM2Json** converts DTSM annotated diagrams into an intermediate description of the topology specified in a JSON file.
- **Json2MC** instantiates the semantics of the topology specified in the DTSM diagram, according to the assumptions described in Section 4.2 and to the model defined in Section 4.2.1, in a Lisp script

(resp., text file) which contains the temporal logic model (resp., the FOL model).

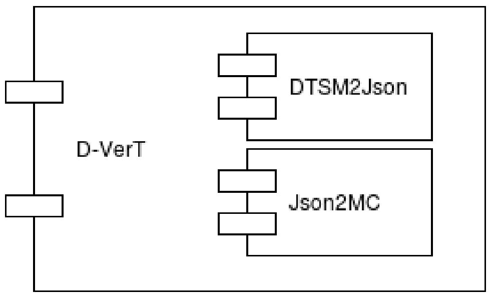


Figure 2: **D-VerT** components.

The activity flow of the components of **D-VerT** is shown in Fig. 3.

The tool, currently available as a standalone application, will be integrated into the DICE IDE in future versions, and the user will be able to access its functionalities via the DICE GUI. It will be possible to graphically define the DTSM model, configure it, annotate it with the desired properties and run the needed verification tasks. As already mentioned, the outcome of such tasks will be then displayed directly in the IDE, in order to make the underlying tools transparent to the user, who will not directly access and manipulate them, and to ensure a uniform user experience.

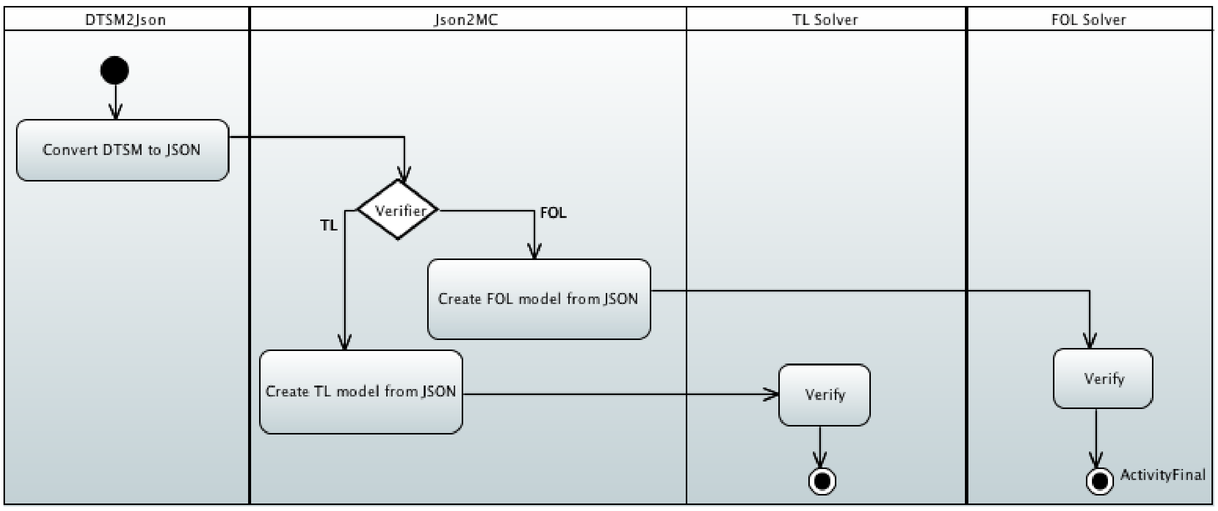


Figure 3: Activity diagram for **D-VerT**.

3.1 Model-checking tools

Three external tools performs verification of DTSM diagrams.

- **Zot**[5] is a bounded model checker. It supports different logic languages: its core uses CLTL and on top of it a fragment of the TRIO metric temporal logic. **Zot** supports many encodings of temporal logic as SMT problems by means of plug-ins. It offers three usage modalities. *Bounded satisfiability checking* is the one used in the DICE framework: Given a temporal logic formula, the tool returns either an execution trace of the specified system satisfying it, or *unsat*, that is, no counterexample has been found.
- **MCMT**[6] is an infinite-state model checker for checking safety properties of systems manipulating array variables whose size is *a priori* unbounded. It provides a declarative way to define array-based systems such as parametrised, timed, and distributed systems which can then undergo verification of a safety problem; more precisely, an **MCMT** model comprises:

[5]`https://github.com/fm-polimi/zot`
[6]`http://users.mat.unimi.it/users/ghilardi/mcmt/`

  - a set of FOL formulae specifying the transitions of the system and its initial configuration;
  - a formula that captures the set of "unsafe" (i.e., undesired) states.
- **Cubicle**[7] is similar to **MCMT**, but it implements a *parallel* symbolic backward reachability procedure to determine whether the unsafe states are reachable or not.

All the above tools rely on Satisfiability Modulo Theories (SMT) [6] solvers to carry out the verification task. In the current setting, **Zot** invokes **Microsoft Z3**[8], **MCMT** uses **Yices**[9], whereas **Cubicle** uses its own SMT solver which is an extension of **Alt-Ergo**[10]. The input language of SMT solvers is standard; all solvers accept input files which conform to the SMT-LIB 2.0 [7] standard.

Fig. 4 shows the four-layered structure of the verification workflow, where **D-VerT** resides in an intermediate level between the annotated DTSM and the SMT solvers. While the former layer provides the input models and receives the verification results, the latter is invoked after the M2M transformation to run the verification task, whose outcome is then processed back to the DTSM layer.

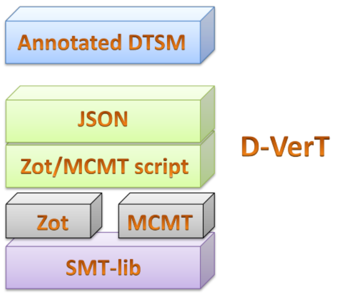


Figure 4: Four-layered structure and **D-VerT**.

[7]`http://cubicle.lri.fr/`
[8]`https://github.com/Z3Prover/z3`
[9]`http://yices.csl.sri.com/`
[10]`http://alt-ergo.lri.fr/`

4 Modeling Data-intensive applications

4.1 Reference technology models

Most of the streaming and batch reference technologies can describe applications by means of topologies, i.e., graphs representing the computation performed by the application. A topology is composed of all the operations performed in the application algorithm, and it defines which steps can be simultaneously executed and which are necessarily sequential. The definition of a topology refers to the design phase and it is part of the outcome of a model-to-model transformation at DTSM level (R2.9, R2.11, R2.12, etc.). The 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 are sources of information and they do not have any role in the definition of the logic of the application. They can be described through the information related to the stream of data they inject in the topology, and the features of their computation is not relevant to define the final outcome of the topology. Computational nodes elaborate input data and produce results which, in turn, are emitted towards other nodes of the topology. A computational node may represent a thread running on a virtual machine, or even a complex system consisting of a DIA itself. Assumptions and restrictions that are currently adopted at this stage of the project are detailed later. Non-functional properties for such nodes are, for instance, the size of data they produce, the emitting rate of data (or its stochastic distribution along with its average value), the failure rate and so on. Finally, a topology defines exactly the connections among the nodes which allow the communication based on message exchange. Therefore, for any node n, it is statically defined at design time (DTSM model) both the list of nodes subscribing to n - i.e., receiving its emitted messages - and the list of nodes n is subscribed to.

While in some cases topology graphs can be derived from the application defined in a declarative way (e.g. in Apache Spark and Apache Flink), other technologies, such as Apache Storm and Apache Samza, allow users to directly specify the topology. Therefore, they let users define applications in a compositional way by combining multiple blocks of computation. The rationale behind Big Data frameworks (pure streaming or batch) found in DIAs may vary from case to case and a unifying approach has not been adopted yet, although the coexistence of different approaches is common in many applications. The work described in this document is the result of the analysis of Apache Storm, which is considered as a reference technology as most of the concepts can be adapted to other frameworks, since the model is very general.

We remark that the design of the verification tool and the tool itself are independent from the specific Big Data framework used (Apache Storm at this stage).

4.1.1 Terminology

- *Spout*: streaming sources which input data into the topology.
- *Bolt*: processing components of the application.
- *Streams*: edges connecting spouts and bolts; they define how data flows into the application. Streams are always point-to-point connections between a sender node (spout or bolt) and a receiver node (always a bolt).
- *Tuple*: atomic data emitted or received by spouts and bolts.
- *Ack*: confirmation message (standing for *acknowledgment*) notified by a bolt after processing an input tuple. Acking is active when the topology is set to be reliable. In such topologies, tuples are associated with a timeout before which they must be acknowledged, otherwise they are newly emitted by the spouts.
- *Failure*: temporary or permanent absence of functionality of a bolt. A failure may cause spouts to newly emit the original tuples.

4.2 Modeling assumptions and topology model

The topology model is constructed under the following assumptions:

- Spouts[11] do not have any input queue or incoming connection from other computation nodes, whereas bolts have an internal queue which stores the incoming tuples received from subscribed nodes of the topology.
- Deployment details, such as, for instance, the number of worker processes and the underlying architecture, are not considered.[12]
- All the queues have unbounded size, which is represented by means of a positive integer value.
- The size of tuples is not relevant and is assumed constant for all tuples.
- Each bolt has the following parameters:
  - σ is a positive real value which abstracts the functionality of a bolt. It expresses a ratio between the size of the input stream and the outgoing stream. For instance, if a bolt filters tuples, i.e., the number of incoming tuples is greater than the number of outgoing tuples, then $0 < \sigma < 1$.
  - $Lenmax$ is the maximum allowed size of the queue.
  - $Takemax$ is the maximum number of concurrent threads that may be instantiated in a bolt. Therefore, an active bolt can remove $Takemax$ tuples from its queue at the same time.
  - α is a positive real value which represents the exact amount of time that a bolt requires to elaborate one tuple.
  - $rebootTime$ (min/max) is a positive real value which represents the amount of time that a bolt requires to restore its functionality after a failure.
  - $idleTime$ (max) is a positive real value which represents the amount of time that a bolt may be idle.
  - $timeToFailure$ (min) is a positive real value which represents the amount of time between two consecutive failures.
- Each spout can emit concurrently at most E number of tuples.
- The behavior of a bolt is defined by a sequence of five different states $idle, execute, take, emit, fail$ with the following meaning:
  - *idle*: no tuples are currently processed in the bolt.
  - *execute*: at least one, and at most $Takemax$ tuples, are currently elaborated in the bolt.
  - *emit*: the bolt emits tuple towards all the bolts that are subscribed.
  - *take*: 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.
  - *fail*: the bolt is currently failed.
- Failures of bolts are promptly recovered locally, i.e., the topology does not have an internal state to represent the anomaly except the internal state of the node. Moreover, failures are independent from each other.

Figure 5 shows the automaton defining the behaviour of a bolt.

4.2.1 Topology model

This section provides an informal description of the model, called *counter networks*, which abstract the behaviour of a topology. The functional part of a topology, which consists of the algorithm implemented by the topology itself, is not captured by the model and does not represent the intent of the verification.

Spouts have only two states, respectively, $idle$ and $emit$, which occur alternatively and not simultaneously. When a spout is in state $emit$ an emit action occurs (two or more emit actions may occur consecutively). A spout has a constant emitting rate which determines the number of emit actions per

[11] Spouts might be reading from queuing brokers such as Kafka, RabbitMQ or Kestrel, or from other sources such as Twitter streaming APIs, but the model represents them as emitting nodes with no queue or storage capability.

[12] A worker process executes a subset of a topology and may run one or more executors for one or more components (spouts or bolts) of this topology. The model does not consider how topologies are implemented into workers and spouts and bolts run within an environment which is not represented.

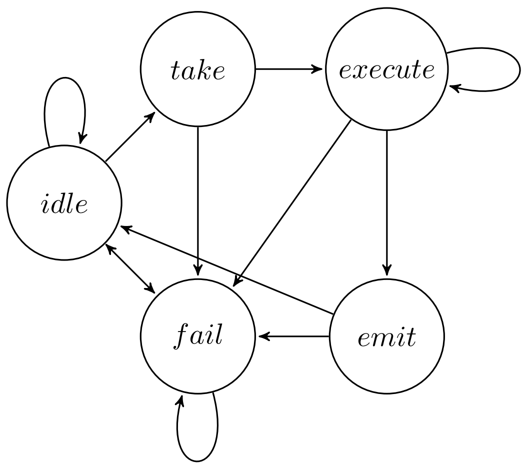


Figure 5: Finite state automaton describing the states of a bolt.

time unit. For each emit action a spout may emit from 1 to a maximum of e tuples. Each *emit* increments the variable representing the size of all the queues of bolts that have subscribed the emitting spout with the number of tuples actually emitted. If a node subscribes to more than one node (spouts or bolts) which emit at the same time, then the size of the queue of that node is incremented accordingly with the sum of all the emitted tuples.

An active bolt in state $take$ extracts tuples from the queue, i.e., when a *take* action is performed. The number of extracted tuples depends on the parallelism level determined by value $Takemax$, which can be different for each bolt. When a *take* occurs, the variable representing the size of the queue of a bolt is decremented by $Takemax$ if the size of the queue is greater than $Takemax$, otherwise it is set to 0. After that, the bolt executes a concurrent processing of the tuples removed from its queue, which lasts α time units. Finally, α time units later, bolts always emit at most one output tuple depending on the parameter σ. An *emit* action may not even produce an output, as the ratio σ determines the relationship between the number of tuples received in input and the number of tuples produced in output according to the relation $\sigma = \frac{n_{out}}{n_{in}}$. The value of σ is either estimated by monitoring an already deployed application, or it is defined according to the functionality implemented by the node. Therefore, a bolt generates one tuple in output only when it concludes the processing of $\sigma \cdot n_{in}$ tuples taken from the queue. Bolts are active components that cannot stay in state *idle* longer than a maximum delay if their queue is not empty. Therefore, the elapsed time between either the instant when the queue becomes non-empty or a bolt emits (and its queue is non-empty) and the following *take* action is never greater that $idleTime$.

Bolts may fail, whereas spout failures are not considered in the model. A failure may occur in any moment and, when it occurs, all tuples in the queue at that moment are lost. The queue occupancy is then immediately set to 0 and, in the case of a reliable topology, the spouts that are ancestors of the failed bolt emit a new set of tuples. The size of the set is determined by the values σ of the intermediate bolts between the spout and the failed bolt. The *time-to-failure* metric, in the current model, is representative of the minimum time between two consecutive failures.

Bolts behave according to the automaton depicted in Fig. 5.

4.3 Temporal logic model

The temporal logic model is written in Constraint LTL over clocks (CLTLoc) enriched with discrete counters, an extension of LTL allowing arithmetical variables to occur in atomic formulae. More precisely, the logic allows for two kinds of atomic formulae:

- atomic formulae over $(\mathbb{R}, \{<, =\})$ contain arithmetical variables which behave as clocks of Timed Automata (TA). For instance, a possible atomic formula over clock x is $x < 4$, where $x \in \mathbb{R}$.
- atomic formulae over $(\mathbb{N}, \{<, =\}, +, \cdot)$ contain arithmetical variables without any semantic restriction. For instance, an atomic formula of this second kind is $x + y < 4$, where both x and y are in

$\mathbb{N}$.

A clock x measures the time elapsed since the last"reset" of x, which occurs when $x = 0$. Its value can be compared with constants in constraints of the form $x \sim c$, where c is a constant value in $\mathbb{N}$. A counter y stores a quantitative value and can be incremented, decremented and tested against a constant value.

Let X be a finite set of clock variables x over $\mathbb{R}$, Y be a finite set of variables over $\mathbb{N}$ and AP be a finite set of atomic propositions p. CLTLoc formulae with counters are defined as follows:

$$\phi := p \mid x \sim c \mid y \sim c \mid \mathrm{X}y \sim z \pm c \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi$$

where $x \in X$, $y, z \in Y$, $c \in \mathbb{N}$ and $\sim \in \{<, =\}$, **X**, **Y**, **U** and **S** are the usual "next", "previous", "until" and "since" operators of LTL [1]. An *interpretation* of a formula is a mapping associating every variable $v \in X \cup Y$ (resp. constant c) with a value in $\mathbb{R}$ (resp., $\mathbb{N}$), plus a mapping associating each element i of $\mathbb{N}$ with the subset of propositional letters of AP that hold at that position. The semantics of CLTLoc is defined as for LTL except for formulae $x \sim c$ and $\mathrm{X}y \sim z \pm c$. Intuitively, formula $x \sim c$ states that the value of clock x is in relation $\sim$ with c, and formula $\mathrm{X}y \sim z \pm c$ states that the next value of variable y is in relation $\sim$ with $z \pm c$.

The satisfiability of CLTLoc formulae can be computed through the Bounded Satisfiability Checking (BSC) technique implemented in a plugin of the **Zot** tool called **ae^{2}zot** (see Section 3, Fig. 2).

The temporal model defines, through a set of CLTLoc formulae, the behaviour of arbitrary topology structures, with particular reference to the Apache Storm technology. It defines, for each node, the order of the actions, the increment/decrement of the queue occupancy, and also additional temporal constraints concerning the time spent in each state. In this way, when the model to be fed to the tool described in Section 5 is created, it is possible to configure both the overall structure of the model and several quality characteristics.

In the next sections we list three formulae exemplifying the possible types of constraints that the model includes.

4.3.1 State machine

The following formula (where $\Rightarrow$ is the entailment connective and `orig` marks the initial element 0 of the model) specifies the condition for *processing*, which represents the state of a bolt in which the take, execute and emit actions can be performed. If a bolt is processing tuples, then state `process` holds until the next emit action or until the instant when a failure occurs; in addition, the state holds since the last take.

$$(\mathtt{process} \Rightarrow \mathtt{process}\,\mathbf{S}\,(\mathtt{take} \vee (\mathtt{orig} \wedge \mathtt{process})) \wedge \mathtt{process}\,\mathbf{U}\,(\mathtt{emit} \vee \mathtt{fail}) \wedge \neg\mathtt{fail})$$

4.3.2 Queue occupancy

The size of the queue of a bolt is measured by a positive integer variable q. The value of q is incremented each time other nodes in the topology send tuples to the bolt, whereas it is decremented when a take action occurs. To model the arrival of tuples in input, proposition `add` becomes true when some of the nodes to which the bolt subscribes emit new tuples. In the first of the next two formulae, variable $r_{\mathtt{add}}$ is the number of incoming tuples that a bolt receives when `add` holds. Therefore, when `add` holds, but no take and failures occur at that time, i.e., $\neg\mathtt{take} \wedge \neg\mathtt{startFailure}$, then the value of queue in the next position $\mathrm{X}q$ is updated with the current value q incremented with $r_{\mathtt{add}}$. The second formula defines the effect of a take action on the next value of q which is determined by the number of incoming tuples that a bolt receives at that moment, when take occurs, and the number of tuples that are removed from the queue to initiate a new processing phase (`process`).

$$\mathtt{add} \wedge \neg\mathtt{take} \wedge \neg\mathtt{startFailure} \Rightarrow (\mathrm{X}q = q + r_{\mathtt{add}}))$$
$$\mathtt{take} \Rightarrow (\mathrm{X}q = q + r_{\mathtt{add}} - r_{\mathtt{process}})$$

4.3.3 Timing constraints

To measure the time delay between events we use, for each bolt and spout, a pair of clocks which are reset alternatively; nevertheless, for the sake of conciseness, the next formulae use a shorthand $t_{\texttt{phase}}$ to indicate the clock of this pair that is relevant in the current instant. Clocks are reset when the associated monitored event occurs and are tested afterwards, to verify if the delay measured by the clock satisfies a certain bound. The first formula below defines the condition for resetting clock $t_{\texttt{phase}}$: this happens when either a take or a failure occurs, or when the bolt becomes idle. The second formula imposes that when an emit action occurs, then the duration of the current processing phase is between $\alpha - \epsilon$ and $\alpha + \epsilon$, for a certain positive value ϵ.

$$(t_{\texttt{phase}} = 0) \iff (\texttt{orig} \vee \texttt{take} \vee (\texttt{fail} \wedge \neg \mathbf{Y}(\texttt{fail})) \vee (\texttt{idle} \wedge \neg \mathbf{Y}(\texttt{idle})))$$

$$\texttt{process} \wedge \texttt{emit} \Rightarrow (t_{\texttt{phase}} \geq \alpha - \epsilon) \wedge (t_{\texttt{phase}} \leq \alpha + \epsilon)$$

Section 5 describes the component of **D-VerT** that produces an instance of the entire set of formulae modelling a topology.

4.3.4 Tool modification

To deal with both discrete counters and clocks in the same CLTLoc specification, as done in the model of Apache Storm topologies presented above, the implementation of the **ae^{2}zot** plugin that is capable of handling CLTLoc formulae must be suitably modified. In fact, clocks and discrete counters obey different kinds of semantic constraints; for example, whereas all clocks, by definition, advance of the same quantity (i.e., time changes is a uniform way), the discrete counters evolve independently of one another. This section briefly hints at the extensions implemented in the **ae^{2}zot** plugin to take into account these differences.

First, the method which introduces the semantics constraints on clock variables is modified so as to avoid defining such constraints for discrete counters. To this end, the interface of the method is changed from

```
(defun gen-regions (bound discrete-regions parametric-regions) ... )
```

to

```
(defun gen-regions (bound discrete-regions parametric-regions
                     discrete-counters) ... )
```

to introduce a new parameter, `discrete-counters`, which indicates the list of variables that are counters over $\mathbb{N}$. Then, the body of the method is modified so as to define temporal constraints for all variables appearing in the CLTLoc formulae, except for those occurring in list `discrete-counters`.

In addition, the introduction of discrete counters increases the expressive power of CLTLoc and makes the logic in general undecidable, so that the procedure implemented in the **ae^{2}zot** for determining the satisfiability of a CLTLoc formula might not terminate. Nevertheless, in a best-effort approach, we can introduce suitable constraints to guarantee the soundness of the result when the procedure terminates (i.e., to guarantee that a model actually exists when the procedure finds one). More precisely, as customary in Bounded Satisfiability Checking procedures, we limit the search for models satisfying the given CLTLoc formula to periodic ones of the form $\alpha(s\beta)^\omega$; that is, the procedure looks for models where a suffix $s\beta$ of finite length is repeated infinitely many times after a prefix α. In practice, given a CLTLoc formula, the **ae^{2}zot** plugin tries to build a partial model $\alpha s \beta s$ of finite length $k+1$, which is representative of the infinite one where suffix $s\beta$ is repeated. To this end, **ae^{2}zot** imposes suitable constraints at the two special positions $|\alpha s| = i_{loop}$ and $|\alpha s \beta s| = k+1$. The tool currently imposes a restrictive set of constraints, which we will look to relax in future works within Task T3.3. The constraints on positions i_{loop} and $k+1$ that are considered in the procedure to solve the satisfiability of CLTLoc formulae are defined in a new method, called `gen-periodic-arith-terms`. In the following snippet, `term` is an element from the list `periodic-arith-terms` containing the variables for which those extra constraints are required.

```
(defun gen-periodic-arith-terms (periodic-arith-terms)
    (if periodic-arith-terms in periodic-arith-terms
        (loop for term in periodic-arith-terms
        collect
             `(<= ,(call *PROPS* term (the-iloop))
                  ,(call *PROPS* term (1+ (kripke-k *PROPS*)))))))
```

Finally, the interface of function `zot` which is invoked in the **ae^{2}zot** plugin to launch the decision procedure is modified to allow the caller to specify the list of discrete counters appearing in the formula through parameter `discrete-counters`, and the set of counters which require specific constraints, through parameter `periodic-terms`.

```
(defun zot (the-time spec
    &key
    ...
    (smt-solver :z3) (logic :QF_UFIDL) (smt-assumptions nil) ...
    (periodic-terms nil)
    (smt-lib :smt)
    (over-clocks 0)
    ...
    (parametric-regions nil)
    (discrete-counters nil)
    ...
  )
```

4.4 First Order Logic model

To complement the temporal logic-based formal semantics of DIAs presented in Section 4.3, an alternative one, given in terms of so-called *array-based systems* has also been defined. Array-based systems allow users to describe applications that are timed, distributed, and parametrised in the number of processes. As such, a model of DIAs based on the formalism of array-based systems can complement a temporal logic one for 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 bolts.

Array-based systems can be formally verified through a decision procedure based on *backward reachability*. Backward reachability analysis is based on the idea of repeatedly computing the pre-image of the set of unsafe states (obtained by complementing the property to be verified) and checking for fix-point and emptiness of the intersection with the set of initial states. This technique can be used to analyse parametrised systems consisting of a finite (but unknown) number n of identical processes modelled as extended finite state automata, which manipulate variables whose domains can be unbounded, like integers. The *challenge* for parametrised systems in general, and for DIAs in particular, which we want to tackle in the context of DICE is to check safety properties for any number n of processes.

The specification of an array-based system composed of one array variable a and one transition τ consists of:

- 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 a 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 behaviour of an array-based system, we characterized the set of initial states of the system and the action ordering in the system by a set of transitions. Both the initial state and the transitions introduce timing constraints for the time spent in each state.

Example of a FOL model. A model of Apache Storm applications should describe the elements of the topology (spouts, bolts, how bolts are subscribed to spouts and among them, queues associated with bolts) as well as their behaviour. For example, at time stamp $t = 0$ the system is in the idle state, that means all spouts and bolts are in the $(I)dle$ state, the length of the queue associated with each bolt, the number of tuples processed by each bolt, and the time a spout emits, are all 0. As the time elapses, the system state should evolve, meaning that the spouts and bolts should change their state according to Figure 5. The queues should receive tuples from all the spouts and bolts that it is subscribed to and the tuples processed by each bolt should be computed correctly. Spouts and bolts failures should also be considered.

We started modelling a simple Storm application composed by n replicas of a topology consisting of one spout and one bolt. The model is compliant with the terminology and the assumptions introduced in Section 4.1.1 and Section 4.2, respectively, but it does not capture the failures of spouts and bolts. The state of the spout x is indicated by a variable $Spout(x)$. A spout can be in one of the two states: $(E)mit$ or $(I)dle$. The state of the bolt x is indicated by a variable $Bolt(x)$. A bolt can be in one of the four states: $(I)dle$ (the bolt is not emitting, nor taking, nor executing), $(E)mit$ (it emits tuples to the bolts which are subscribed to it), $Ta(K)e$ (it takes tuples from the queue associated to the bolt in order to be processed), $E(X)ecute$ (it performs certain operations with the tuples previously taken from the queue). We also maintain some other variables:

- $L(x)$: the length of the queue associated with the bolt x
- $P(x)$: the number of tuples that were processed by the process x in the bolt B since the last $Ta(K)e$
- $s_{time}(x)$: the time elapsed since the last emission of spout x; the distance between consecutive emissions of a spout is at least T^{spout}_{min} time units and at most T^{spout}_{max} time units; after T^{spout}_{max} time units elapse, another process can operate on the spout.

The initial state of the system is described by the following formula:

$$t = 0 \land \underset{x}{\forall} (S(x), B(x) = I \land L(x), P(x), s_{time}(x) = 0)$$

meaning that initially the clock is set to 0, the spouts and the bolts are in the $(I)dle$ state, the length of the queues, the number of tuples processed and the value of s_{time} are all 0, *for all* processes x.

The formal description of the topology composed of n replicas of one spout and one bolt is given by the transition system in Appendix A. We describe each transition by a logical formula that corresponds to guarded assignment systems, relating 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 instance, in the transition (1) the subformula: $c > 0 \land \ldots \land flag = 0$ represents the *guard* (precondition) for the state variables to be *updated* $t' = t + c \land \ldots \land canTimeElapse' = 1$

$$\underset{x,c}{\exists}\, c > 0 \land S(x) = E \land T^{spout}_{min} < s_{time}(x) + c < T^{spout}_{max} \land flag = 0 \land$$
$$\underset{j}{\forall}\left(\begin{array}{lll} t' = & t + c & \land \\ P'(j) = & \text{if } (B(j) = X \text{ and } P(j) - Execrate * c \geq 0) & \\ & \text{then } P(j) - Execrate * c \text{ else } 0 & \land \\ s'_{time}(j) = & \text{if } j = x \text{ then } 0 \text{ else } s_{time}(j) + c & \\ flag' = & 1 & \land \\ statechange' = & 1 & \land \\ canTimeElapse' = & 1 & \end{array}\right) \quad (1)$$

Note that the updates can be seen to model a broadcast action in a parametrised system. For example, in the transition below a process x may determine that $S(x)$ changes nondeterministically to state E or I, while all the other processes do not react by changing their control location in S.

$$
\exists_{x,y} \; statechange = 1 \land flag = 0 \land
$$
$$
\forall_{j} \left(\begin{array}{lll}
statechange' = & 0 & \land \\
S'(j) = & \text{if } j = x \text{ then } (E \text{ or } I) \text{ else } S(j) & \land \\
B'(j) = & \text{if } (j = y \text{ and } B(j) = E) \text{ then } (I \text{ or } K) \text{ else } B(j) & \\
 & \text{elseif } (j = y \text{ and } B(j) = I) \text{ then } K \text{ else } B(j) & \land \\
canTimeElapse' = & 1 &
\end{array} \right)
$$

Our model should ensure that, for all processes, the length of the queue associated with a bolt does not exceed the maximum length $Lenmax$. Proving this safety property amounts to checking that states that satisfy the following formula are not reachable, where the formula is the negation of the property we want to check and describes the sets of *unsafe* states:

$$
\exists_{x} \; L(x) > Lenmax
$$

Examples of other interesting safety properties are: if a bolt i emits, then tuples from the corresponding queue will be processed (the bolt will be in the $Ta(K)e$ state); a bolt can not stay in the $(I)dle$ state indefinitely if its queue is not empty, etc. It is also challenging to check different safety properties on the assumptions that bolts are either fast/slow compared to the spouts they are subscribed to.

The array-based system model described above can be implemented and validated in state-of-the-art model checkers (e.g. **MCMT**[13], **Cubicle**[14], **Safari**[15]). We made extensive experiments with **MCMT** and **Cubicle**. The results show that they both can be used for our purposes; however the input language of **Cubicle** is simpler, hence the translation activity that must be performed by **D-VerT** from the JSON format to the input language of the tool is simpler. Moreover, **Cubicle** allows for the definition of matrices, which can be a suitable abstraction for modelling systems with m spouts and n bolts. Simple arrays could be used also for this purpose, but they lead to an exponential growth (in the number of spouts and bolts) of the model and no state-of-the-art tool could be used to perform verification on such a large model.

The current work in this line of research focusses on modelling Apache Storm topologies with m spouts and n bolts using matrices.

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

[14] `http://cubicle.lri.fr/`

[15] `http://verify.inf.usi.ch/content/safari`

5 Verification in DICE

This section describes the implementation of component **Json2MC** of Fig. 2.

In the current version of the **D-VerT**tool, we focused on an intermediate step of the complete verification workflow described in Section 3; that is, from the description of topologies in a JSON format to the logical model. The component generates, from the JSON representation of the topology and a model template, the appropriate instance of formal model (in the form of a lisp script), ready to be fed to the selected external model checking tool.

5.1 Architecture and implementation details

Json2MC has been designed to be extensible and configurable. It is composed of a core component, *Model Configurator*, and a set of *model templates*, which embed the syntax and semantics of the different models that have to be produced to run formal verification. As depicted in Fig. 6, *Model Configurator* reads the topology description encoded in JSON format and instantiates the appropriate formal model by rendering the selected template, according to the input configuration.

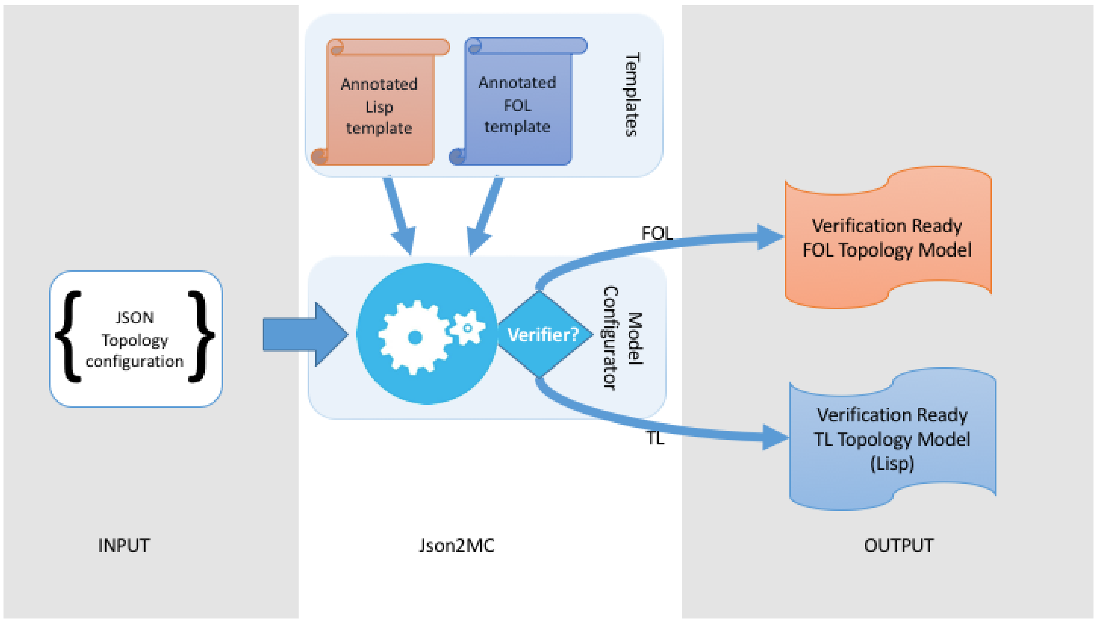

Annotated Lisp template
Annotated FOL template
Templates
JSON Topology configuration
Verifier?
Model Configurator
FOL
TL
Verification Ready FOL Topology Model
Verification Ready TL Topology Model (Lisp)
INPUT
Json2MC
OUTPUT

Figure 6: **Json2MC** translating from JSON to verification-ready model files.

We now provide a brief description of the Model configurator and of the templates; we also show some details about the Lisp macros mechanisms that we used to generate all the needed model formulae in a flexible way.

5.1.1 Model Configurator

This component is implemented in Python, and makes use of the Jinja2[16] library, an open source templating engine mainly used in web programming. In order to produce the desired output model, it requires the definition of a template and a JSON configuration object (or context, in the Jinja2 jargon).

5.1.2 Model Templates

Templates are generally simple text files and they do not require a specific extension. Each template contains variables and expressions, that will be replaced after the rendering process by the values expressed

[16] `jinja.pocoo.org/docs/latest/`

in the configuration file, as well as tags, which allow more complex logics such as conditional statements and filters.

Listing 1 shows a fragment of the temporal logic template file written in Common Lisp and containing variables and tags (all included in double curly brackets). The corresponding output text (this time containing only Lisp code), produced from it, is presented in Listing 2.

Listing 1: Template fragment representing the topology configuration.

```
...
;TOPOLOGY DEFINITION
(defconstant the-spouts '({{ topology.spouts|join(' ', attribute='id') }}))
(defconstant the-bolts '({{ topology.bolts|join(' ', attribute='id') }}))

{% for b in topology.bolts %}
(setf (gethash '{{b.id}} the-topology-table) '({{b.subs | join(' ')}}))
{%endfor%}
...
```

Listing 2: Piece of code rendered from the template in Listing 1.

```
...
;TOPOLOGY DEFINITION
(defconstant the-spouts '(S1 S2))
(defconstant the-bolts '(B1 B2 B3))

(setf (gethash 'B1 the-topology-table) '(S1))
(setf (gethash 'B2 the-topology-table) '(S1 S2))
(setf (gethash 'B3 the-topology-table) '(B1 B2))
...
```

5.1.3 Lisp Formulae expansion

In the case of the temporal logic model we devised a *double templating* layer. In fact, the model configurator takes care only of the general settings of the model (topology structure, single nodes parameters, selected **Zot** plugin). According to this configuration, the logical formulae are then generated by means of Lisp's *macro-expansion* mechanism. In this way, we were able to write the logical formulae only once, and replicate them for the needed number of times based on the topology structure.

Listing 3: Macro defining the processing state of the bolt as described in section 4.3.1.

```
...
(defmacro singleBoltsBehaviour(bolts)
`(&&
    ,@(nconc
        (loop for i in bolts collect
        `(->
            (-P- ,(format nil "PROCESS_~S" i))
            (&&
                (!! (-P- ,(format nil "FAIL_~S" i)))
                (since
                    (-P- ,(format nil "PROCESS_~S" i))
                    (||
                        (-P- ,(format nil "TAKE_~S" i))
                        (&& orig (-P- ,(format nil "PROCESS_~S" i)))))
                    (until
                        (-P- ,(format nil "PROCESS_~S" i))
                        (||
                            (-P- ,(format nil "EMIT_~S" i))
                            (-P- ,(format nil "FAIL_~S" i)))))))
```

```
...
```

We can see from Listing 4 how the macro shown in Listing 3 is expanded to apply the formula to the three bolts B1, B2 and B3.

Listing 4: Formulae generated by expanding the macro in Listing 3.

```
...
(&&
    (-> (-P- "PROCESS_B1")
    (&&
         (!! (-P- "FAIL_B1"))
        (SINCE (-P- "PROCESS_B1") (|| (-P- "TAKE_B1") (&& ORIG (-P- "
           PROCESS_B1"))))
        (UNTIL (-P- "PROCESS_B1") (|| (-P- "EMIT_B1") (-P- "FAIL_B1")))))
    (-> (-P- "PROCESS_B2")
    (&&
        (!! (-P- "FAIL_B2"))
        (SINCE (-P- "PROCESS_B2") (|| (-P- "TAKE_B2") (&& ORIG (-P- "
           PROCESS_B2")))) (UNTIL (-P- "PROCESS_B2") (|| (-P- "EMIT_B2") (-
           P- "FAIL_B2")))))
    (-> (-P- "PROCESS_B3")
    (&&
        (!! (-P- "FAIL_B3"))
        (SINCE (-P- "PROCESS_B3") (|| (-P- "TAKE_B3") (&& ORIG (-P- "
           PROCESS_B3"))))
        (UNTIL (-P- "PROCESS_B3") (|| (-P- "EMIT_B3") (-P- "FAIL_B3"))))))
...
```

5.2 Verification workflow

Starting from a simple topology example, we now provide an overview of the workflow going from the topology representation to the actual verification activity.

5.2.1 Topology Description

The topology in Fig. 7, is composed of two spouts (S1, S2) and three bolts (B1, B2, B3). The spouts have the same emitting rate, while the bolts have different characteristics. σ_{B1} has value 2.0, meaning that on average, for each input tuple, bolt B1 emits two tuples. This could be the case where a compound tuple (e.g., containing a list of values) is broken into multiple tuples. Bolt B2 has σ_{B2} equal to 0.5, that is, it could perform a join operation on the input tuples coming from the two spouts. Both bolts take on average the same time (α) to process a each tuple, and have different parallelism level. Bolt B3 is faster than the other bolts and has a replication factor of 3.

5.2.2 JSON encoding

The topology presented above, specified through a DTSM diagram, is encoded into a JSON object by the **DTSM2Json** component.
As can be noticed from Listing 5, the JSON format allows us to capture in a compact and readable way all the needed parameters to build the model and run the verification, such as:

- topology-related settings:
  - list of spouts with specific parameters:
    - `emit_rate`: spout average emitting rate
  - list of bolts with specific parameters:

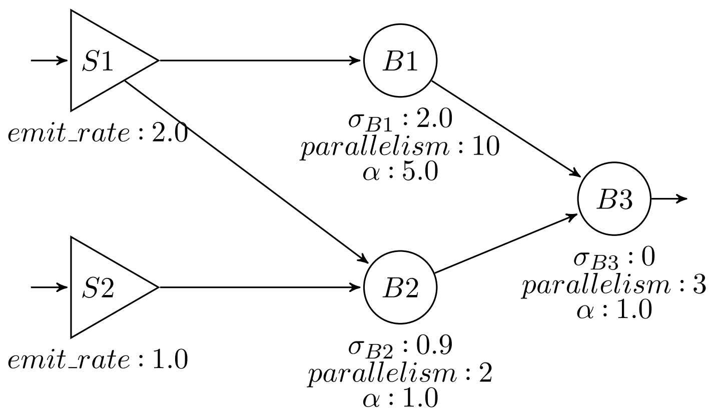


Figure 7: DAG representing a simple topology.

  * `subs`: the subscription list
  * `parallelism`: level of parallelism chosen for each bolt corresponding to the already mentioned $Takemax$ parameter of the model.
  * `alpha`: average processing time for the single tuple (α)
  * `sigma`: operation performed, in terms of output tuples / input tuples ratio (σ)
  * `min_ttf`: minimum time to failure

- structure of the topology, expressed through the combination of the subscription lists of all the bolts

• verification-related settings:

- `num_steps`: number of time instants to be explored in the verification phase
- `periodic_queues`: the queues to be monitored for a periodically increasing trend.
- `queue_threshold`: the maximum level of occupancy that should not be exceeded by any queue.
- `plugin`: **Zot** plugin to be used (**ae^{2}zot** or its variant **ae^{2}bvzot**, which can sometimes be more efficient)

Listing 5: Example JSON file describing a simple topology.

```
{
"app_name": "Simple Topology",
"description": "",
"version": "0.1",
"topology":{
    "spouts":[
        {"id":"S1",
        "avg_emit_rate":2.0},
        {"id":"S2",
        "avg_emit_rate":1.0}
    ],
    "bolts":[
        {"id":       "B1",
        "subs":      ["S1"],
        "alpha":     5.0,
        "sigma":     2.0,
        "min_ttf":   1000,
        "parallelism": 5},
        {"id":       "B2",
        "subs":    ["S1", "S2"],
        "alpha":    5.0,
        "sigma":    0.5,
        "min_ttf":  1000,
        "parallelism": 10},
        {"id":        "B3",
        "subs":     ["B1", "B2"],
        "alpha":    1.0,
        "sigma":    0.0,
        "min_ttf":  1000,
        "parallelism": 3}
        ],
    "min_reboot_time":10,
    "max_reboot_time":100,
    "max_idle_time":  0.01,
    "queue_threshold": 20},
"verification_params":{
    "plugin" : "ae2bvzot",
    "max_time" :  20000,,
    "num_steps":10,
    "periodic_queues":["B1", "B2","B3"]}
}
```

5.2.3 Output trace

The result of the verification task can be a "counter-example" history representing a computation that satisfies the conditions imposed (i.e., that violates the desired property), or the message that the model is unsatisfiable. In all the examples provided in this document, the property we want to verify is that *all the queues are bounded and, for none of them, the level of occupancy exceeds a certain threshold (`queue_threshold`) in the number of time instants considered (`num_steps`).* Therefore, satisfying the model (*sat* result), means finding an execution where at least one of the queues is unbounded and exceeds the given threshold level.

In case of a *sat* outcome, the **Zot** bounded satisfiability checker provides a very raw textual result (see Listing 6), which lists the values of all the model variables in the different time steps (numerical values for discrete counters and clocks, and simply the name of the Boolean variables that are true in each instant).

Listing 6: Fragment of the output trace produced by Zot when the model is satisfied.

```
...
------ time 0 ------
...
------ time 1 ------
STARTIDLE_B1
TAKE_B2
IDLE_B1
PROCESS_B2
R_EMIT_S2 = 0
R_PROCESS_B2 = 2
CLOCK_BF_B1 = 999.0033333333?
Q_B1 = 1
L_EMIT_S1 = 0
Q_B2 = 2
PT_S2_1 = 2.1033333333?
PT_B1_0 = 1.0
PT_B2_0 = 2.0966666666?
R_EMIT_S1 = 0
R_REPLAY_S1 = 0
BUFFER_B3 = 1
```

```
DELTA = 0.0033333333?
R_ADD_B2 = 0
R_ADD_B3 = 0
R_REPLAY_S2 = 0
R_FAILURE_B2S1 = 0
R_EMIT_B2 = 0
PT_S1_0 = 0.8966666666?
R_FAILURE_B2S2 = 0
TOTALTIME = 0.0033333333?
R_PROCESS_B1 = 0
BUFFER_B1 = 0
R_FAILURE_B3S1 = 0
R_EMIT_B1 = 0
Q_B3 = 0
PT_B1_1 = 0.0
CLOCK_BF_B3 = 994.5066666666?
PT_S2_0 = 0.8966666666?
R_FAILURE_B3S2 = 0
PT_B3_0 = 2.0
CLOCK_BF_B2 = 995.1
R_ADD_B1 = 0
R_EMIT_B3 = 0
R_PROCESS_B3 = 0
PT_B2_1 = 0.0
BUFFER_B2 = 0
PT_S1_1 = 2.1033333333?
L_EMIT_S2 = 1
R_FAILURE_B1S1 = 0
PT_B3_1 = 5.1
------ time 2 ------
PROCESS_B1
EMIT_S2
TAKE_B1
ADD_B2
...
------ time 3 ------
...
------ time 4 ------
...
------ time 15 ------
...
------ end ------
```

Since these output traces are not very user-friendly and inspecting them in order to better understand the system behaviour is quite time-consuming, we implemented a new prototype module to present the traces in a graphical way.

For the time being this functionality simply gets the output trace (in case the model is satisfiable) and plots the evolutions of the bolt-related variables over time. Figure 8 provides an example of such output traces. Each of the three plots refers to a specific bolt: queue trends are displayed as solid black line. Green and red solid lines show the processing activity of the bolts, while the dashed lines illustrate the incoming tuples from the subscribed nodes (emit events).

This module exploits the python library `matplotlib.pyplot`[17], a MATLAB-like plotting framework. The plot can be customized via a JSON configuration file that allows users to choose the variables to be plotted together with the line style. Further development is planned in order to let the trace be displayed in the DTSM topology diagram once the verification is complete. A possible way to display the result is by using UML Timing Diagrams.

[17] `http://matplotlib.org/`

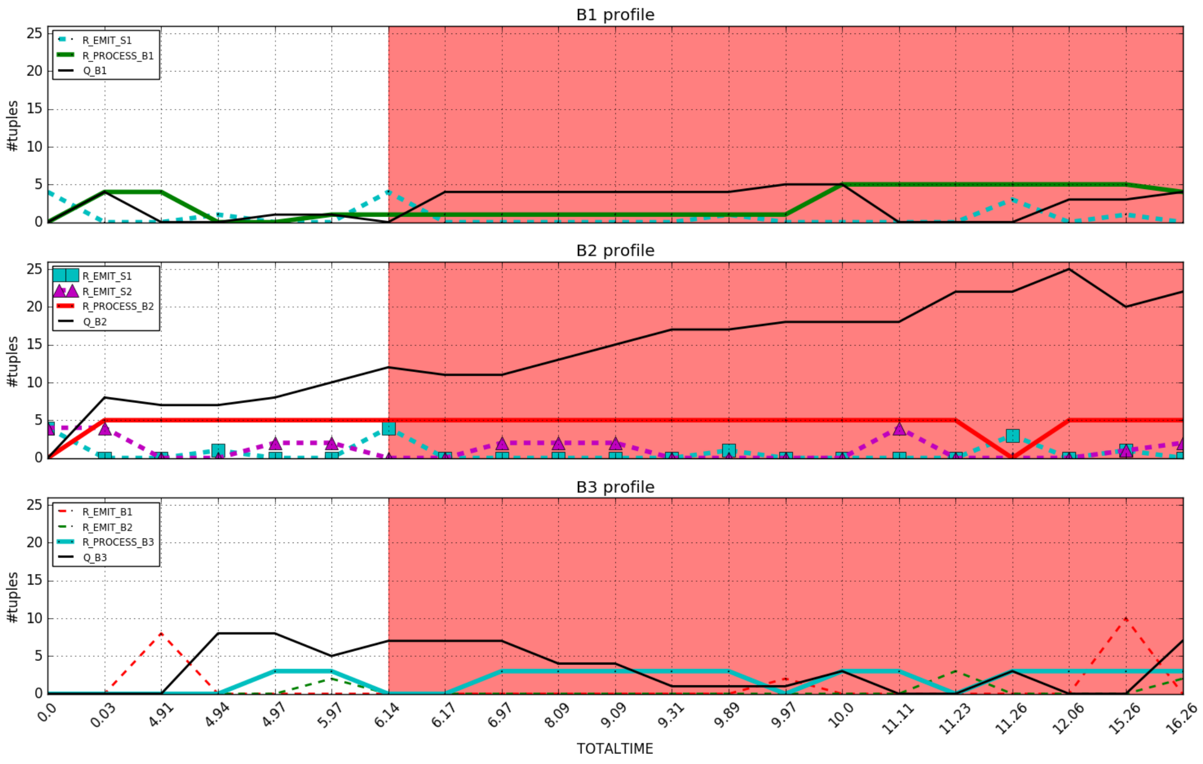


Figure 8: Graphical representation of an output trace.

6 Validation

This section presents the results of the validation activity of the **D-VerT** tool that was performed through a simple use case. This was done by running some verification tasks on the DIA topology presented in Figure 9.

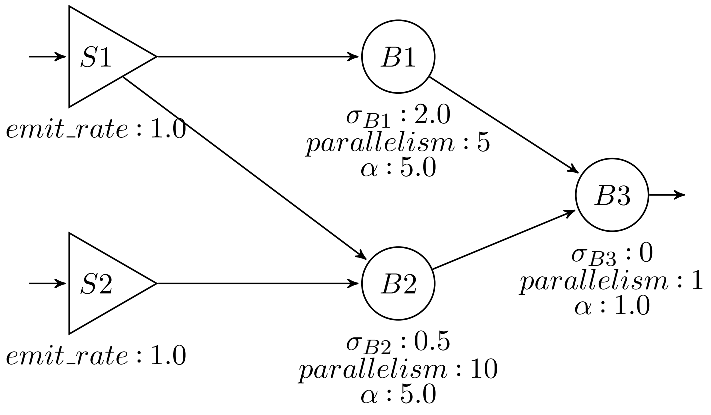


Figure 9: A simple DIA topology.

When running the verification task on the topology configured “as is”, we gathered a counterexample (see Fig. 10), showing an increasing trend in the queue of bolt B3 as can be noticed by looking at the black dotted line on the third plot (B3 profile). So, bolt B3 appears not to be capable of processing the incoming quantity of messages from bolts B1 and B2, which in fact was the expected behaviour.

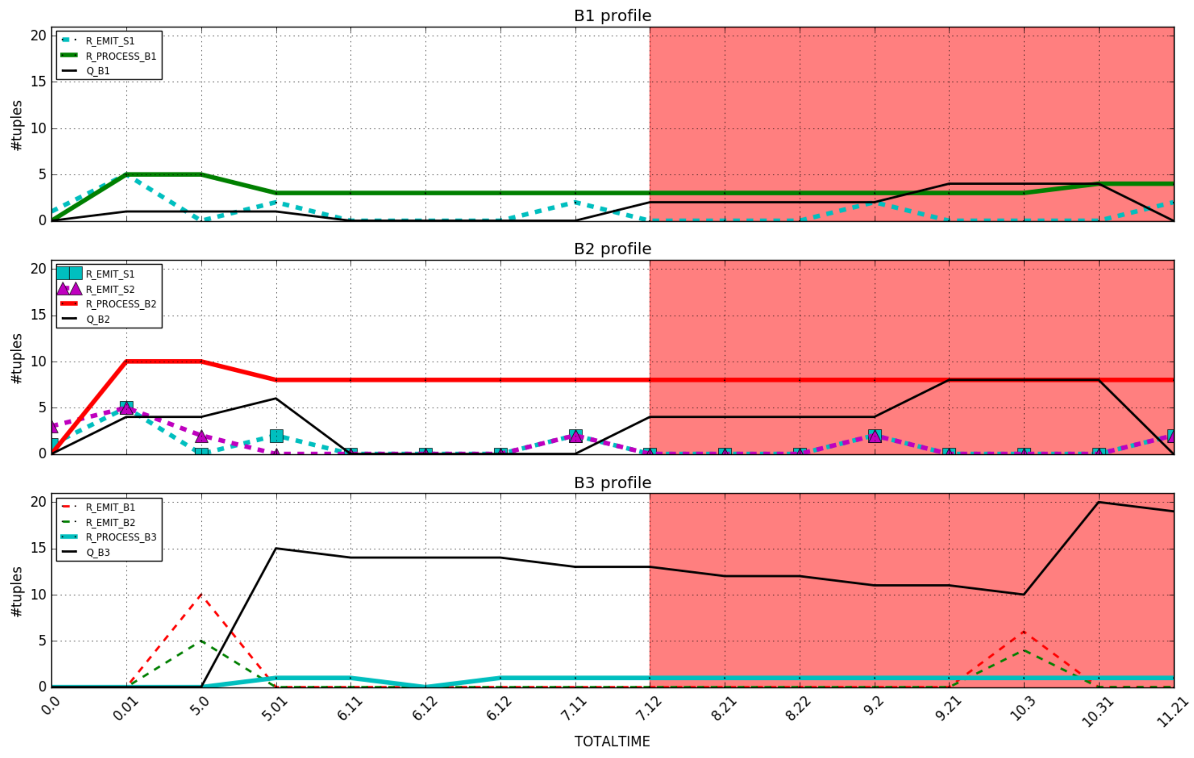


Figure 10: First trace showing increasing queue for bolt B3 ($parallelism_{B3} = 1$).

Assuming that the processing time cannot be improved for the function performed by bolt B3 (i.e., α cannot be lowered) we tried to increase the level of parallelism of the bolt to 3 and rerun the verification (the possibility of increasing the parallelism is, in fact, another assumption). This time bolt B3 seems to better handle the incoming load, even though the profile of its queue has a spike in the end. Another problem is given by the bolt B2, whose queue occupancy level shows a slightly increasing trend.

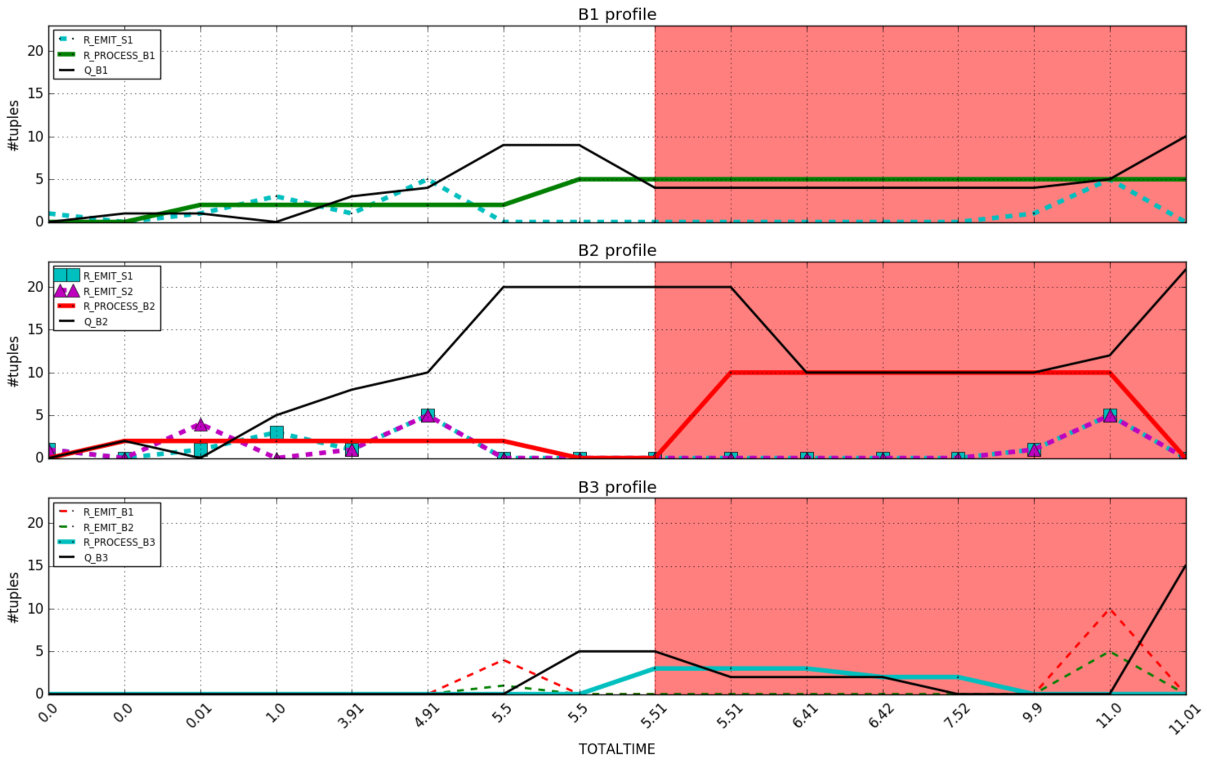


Figure 11: Second trace showing increasing queues for both B3 and B2 ($parallelism_{B3}$ = 3).

Considered this second outcome, we tried to further increase the parallelism of both bolts B2 and B3, respectively to 12 and 5. With this configuration, we finally obtained an *unsat* result, that is, no counterexample violating the properties (which are "*eventually, the queue occupancy is greater than a fixed amount*", and "*the queues cannot decrease indefinitely*") was found by the tool.

We also ran verification tasks on more complex topologies inspired by the case studies that are being considered in the DICE project, in order to understand how the prototype **D-VerT** tool performs on non-trivial scenarios. These experiments showed that, as customary for formal verification techniques, the execution time increases significantly as the size of the analysed topology grows. Part of the future work on the **D-VerT** tool, then, will focus on devising ways to mitigate the negative effects of the so-called state explosion problem when topologies increase in complexity.

7 Conclusions 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. The primary focus of our activities was on developing the abstract models representing the data intensive technologies to be analysed (**R3.12**). We implemented such models in order to run verification tasks on them, and we extended existing external tools to let them support the characteristics of the new models. We designed the models to be configurable and provided a layered structure to facilitate the future integration with the DICE framework by decoupling the verification layer from the DTSM diagrams that are still under completion. Currently our tool can be used as a standalone application and provides a graphical output in order to better understand the output traces returned by the underlying external tool.

Requirement ID	Description	Coverage	To do
R3.1	M2M Transformation	0 %	
R3.2	Taking into account relevant annotations	40 %	New annotations for Spark and Hadoop. Privacy annotations
R3.3	Transformation Rules	0 %	
R3.7	Generation of traces from system model	60 %	Integration in the DICE IDE
R3.10	SLA specification and compliance	30 %	Highlighting violated SLA
R3.12	Modelling abstract level	40 %	New abstraction can be considered for Spark and Hadoop frameworks
R3.15	Verification of temporal safety/privacy properties	40 %	Transformation from UML to internal verification model. Theoretical results on correctness and completeness.
R3IDE.2	Timeout Specification	50 %	Integration in the DICE IDE
R3IDE.4.2	Loading the properties to be verified	40 %	Some relevant properties might still be devised
R3IDE.5	Graphical output	60 %	Integration in the DICE IDE
R3IDE.5.1	Graphical output of erroneous behaviours	60 %	Integration in the DICE IDE

Table 1: Requirement coverage at month 12.

7.1 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 M24.

IDE. Most of the effort needed to complete the requirements will focus on the integration of the tool with the DICE framework. All the functionalities under development (**R3.7, R3IDE.2, R3IDE4.2, R3IDE.5, R3IDE5.1**) need to be made available through the DICE IDE in a transparent way.

R3.1. A key aspect in the integration process is the *model to model transformation* (**R3.1**), that will be addressed by the development of the **DTSM2Json** component of **D-VerT**.

R3.2. Additional work will be devoted to the topic of privacy, in order to tackle the problem in a meaningful way and to support new annotations in our models.

R3.10. Further analysis of SLA's, defined by the designer in the UML models, must be elaborated to define which requirements can be supported in the DICE framework, beside timing constraints and the non-functional parameters listed in Section 4. This will be supported with a deeper analysis of industrial case-studies of DICE partners and of real implemented applications available on-line.

R3.12. Other relevant technological frameworks, i.e., Spark or Hadoop MapReduce, will be considered. The TL-model and FOL-model will be either enriched with, or adapted to these additional technologies. The main aspects to formalize are the management of failures, the definition of topologies and the behaviour of nodes.

R3.15. The next achievements concerning this requirement are:

- Elaborating a new model perspective (for all the technological frameworks) considering deployment information associated with the nodes in a topology. In particular, modelling the internals of a single node (i.e., taking into consideration workers, executors and tasks in a single node) would allow for the definition of an intra-node analysis to complement the inter-node analysis proposed in this document.
- Message brokers are currently not part of the model, although they have a key role in a big-data application. Therefore, addressing their functionalities and timing constraints is relevant for the verification purposes of DICE.
- Further investigations are needed to gain a deeper knowledge of the current model and to get theoretical results on the correctness and completeness of the verification through CLTLoc. In particular, a deep analysis of counter networks, introduced in Section 4, must be completed in order to compare their expressiveness with respect to Timed Petri Nets.
- New properties of interest, beside those addressed so far on the evolution of the queue occupancy, may also be elicited and included in the model.

References

[1] Carlo A. Furia et al. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2012.

[2] Ralph L. Disney and Dieter König. *Queueing Networks: A Survey of Their Random Processes*. Vol. 27. 3. Society for Industrial and Applied Mathematics, 2006, 335–403. 69 pp.

[3] Beatrice Bérard et al. "Comparison of the Expressiveness of Timed Automata and Time Petri Nets". In: *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*. 2005, pp. 211–225.

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

[5] 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.

[6] Clark W Barrett et al. "Satisfiability Modulo Theories." In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2010.

A Details of the Formal Models

A.1 Temporal Logic Model.

A Storm Topology is a directed graph $\mathbf{G} = \{\mathbf{N}, \mathbf{E}\}$ such that the set of nodes $\mathbf{N} = \mathbf{S} \bigcup \mathbf{B}$ includes in the sets of spouts ($\mathbf{S}$) and bolts ($\mathbf{B}$), while the set of edges $\mathbf{E} = \{Sub_{i,j} | i \in \mathbf{B}, j \in \{\mathbf{S} \bigcup \mathbf{B}\}\}$ defines how the nodes are connected each other via the subscription relationship. $(Sub_{i,j})$ is a shortand for "bolt i subscribes to the streams emitted by the spout/bolt j.
Each bolt is represented as a computation node having a receive queue where the tuples to be processed by it are stored. As already highlighted, spouts do not have such queues. Each spout can either emit tuples into the topology or being idle. Bolts can be either in idle, processing or failure state. During the processing phase they read tuples from their receive queues, apply the specific transformation and emit new output tuples.

The main execution steps of each bolt (respectively spout) are: `take()` (respectively `nextTuple()`), `execute()` (respectively `transform()`) and `emit()`. These steps are represented in a more generic way by the propositions:

$\mathtt{take}_i$ indicates that the `take()`/`nextTuple()` step is performed by the i^{th} bolt/spout;

$\mathtt{process}_i$ corresponds to the `execute()`/`transform()` step performed by the i^{th} bolt/spout;

$\mathtt{emit}_i$ indicates that bolt i is emitting tuples;

Single node behaviour

Let `orig` be a shorthand for $\neg \mathbf{Y}(\top)$. The formula is true only in the origin. The behaviour of each single node is defined by the following properties:

$$\bigwedge_{i \in \mathbf{B}} (\mathtt{take}_i \Rightarrow \mathtt{process}_i \wedge \mathbf{X}(\neg \mathtt{take}_i \mathbf{U} (\mathtt{emit}_i \vee \mathtt{fail}_i)) \wedge \neg \mathtt{emit}_i \wedge \mathbf{Y}(\neg \mathtt{process}_i \, \mathbf{S} \, (\mathtt{emit}_i \vee \mathtt{orig} \vee \mathtt{fail}_i))) \tag{2}$$

$$\bigwedge_{i \in \mathbf{B}} (\mathtt{process}_i \Rightarrow \mathtt{process}_i \, \mathbf{S} \, (\mathtt{take}_i \vee (\mathtt{orig} \wedge \mathtt{process}_i)) \wedge \mathtt{process}_i \, \mathbf{U} \, (\mathtt{emit}_i \vee \mathtt{fail}_i) \wedge \neg \mathtt{fail}_i) \tag{3}$$

$$\bigwedge_{i \in \mathbf{B}} (\mathtt{emit}_i \Rightarrow \mathtt{process}_i \wedge (\mathtt{orig} \vee \mathbf{Y}(\neg \mathtt{emit}_i \, \mathbf{S} \, \mathtt{take}_i)) \wedge \mathbf{X}(\neg \mathtt{process}_i \, \mathbf{U} \, \mathtt{take}_i)) \tag{4}$$

$$\bigwedge_{i \in \mathbf{B}} (\mathtt{startIdle}_i \iff ((\mathtt{emit}_i \wedge \mathbf{X}(q_i > 0)) \vee (\neg \mathtt{fail}_i \wedge \neg \mathtt{process}_i \wedge q_i > 0 \wedge \neg \mathbf{Y}(q > 0 \vee \mathtt{emit}_i)))) \tag{5}$$

$$\bigwedge_{i \in \mathbf{B}} (\mathtt{idle}_i \Rightarrow \mathtt{idle}_i \, \mathbf{S} \, \mathtt{startIdle}_j \wedge \mathtt{idle}_i \, \mathbf{U} \, (\mathtt{take}_i \vee \mathtt{fail}_i) \wedge \neg \mathtt{fail}_i) \tag{6}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathbf{G} \, (\mathbf{F} \, (\mathtt{emit}_i))) \tag{7}$$

$$\bigwedge_{i \in \mathbf{B}} (\mathbf{G} \, (\mathbf{F} \, (\mathtt{take}_i))) \tag{8}$$

Node Failure

$$\bigwedge_{j \in B} (\mathtt{startFailure}_j \iff (\neg \mathbf{Y}(\mathtt{fail}_j) \wedge \mathtt{fail}_j))$$

$$\bigwedge_{j \in B} (\mathtt{startFailure}_j \Rightarrow (\bigwedge_{\substack{i \in B \\ i \neq j}} (\neg \mathtt{startFailure}_i)))$$

Single queue behaviour

Each bolt has a receive queue where the incoming tuples are collected before being read and processed by the node. The queues have infinite size and the level of occupation of each j^{th} queue is described by the variable q_j.
In order to express the connections among nodes in the topology, we defined the set of propositions $Sub(j,i)$ where $i \in \{\mathbf{S} \bigcup \mathbf{B}\}$ and $j \in \mathbf{B}$. $Sub(j,i)$ means that "bolt j submits to the stream emitted by spout/bolt i". This allows to establish a direct connection between the emission of a tuple by a node and the arrival of the tuple in the receive queues of the submitting nodes.
We also define $In_j = \{i_0, i_1, \dots, i_n \,|Sub(j,i)\}$ where $j \in \mathbf{B}$ and $i \in \{\mathbf{S} \bigcup \mathbf{B}\}$, as the set of all the spouts/bolts whose streams are subscribed by the bolt j (i.e. the nodes whose tuple are sent to the input queue of the bolt j).
The proposition $\mathtt{add}_j$ is used to indicate that some of the nodes belonging to $In(j)$ are emitting tuples, that is, those tuples are added to the receive queue of bolt j.

$$\bigwedge_{j \in B} (\mathtt{add}_j \iff \bigvee_{\substack{i \in \{S \bigcup B\} \\ Sub(j,i)}} \mathtt{emit}_i)$$

Rates

To represent the quantities of tuples that are taken/processed/emitted by each node in the topology, we introduce the concept of rate defined as "the quantity of tuples considered in the current time unit".
We therefore present the following rates:

- $r_{\mathtt{emit}_j}$: quantity of tuples emitted in the current time unit by task j.
- $r_{\mathtt{take}_j}$: quantity of tuples taken in the current time unit by task j.
- $r_{\mathtt{add}_j}$: quantity of tuples that are added to the receive queue of task j in the current time unit, consisting in the sum of all the tuples currently emitted by the tasks belonging to $In(j)$.

Therefore, the behaviour of the queue given the occurrence of adding and/or receiving event is the following:

$$\bigwedge_{j \in B} q_j \geq 0$$

$$\bigwedge_{j \in B} (\mathtt{add}_j \wedge \neg \mathtt{take}_j \wedge \neg \mathtt{startFailure}_j \Rightarrow (\mathrm{X}q_j = q_j + r_{\mathtt{add}_j}))$$

$$\bigwedge_{j \in B} (\mathtt{take}_j \Rightarrow (\mathrm{X}q_j = q_j + r_{\mathtt{add}_j} - r_{\mathtt{process}_j}))$$

$$\bigwedge_{j \in B} (\mathtt{startFailure}_j \Rightarrow \mathbf{X}(q_j = 0))$$

Necessary conditions:

$$\bigwedge_{j \in B} ((\mathrm{X}q_j > q_j) \Rightarrow \mathtt{add}_j)$$

$$\bigwedge_{j \in B} ((\mathrm{X}q_j < q_j) \Rightarrow \mathtt{take}_j \vee \mathtt{startFailure}_j)$$

Rates Behaviour

The following properties define the behaviour of rates for all bolts, distinguishing, when needed, between final ("leaf" node of the topology graph) and non-final bolt. $final(i)$ is equivalent to $\neg\exists j : Sub(j,i)$:

$$\bigwedge_{j\in B} (r_{\mathtt{add}_j} \geq 0 \land r_{\mathtt{process}_j} \geq 0 \land r_{\mathtt{emit}_j} \geq 0 \land \mathtt{buffer}_j \geq 0 \land r_{\mathtt{replay}_j} \geq 0)$$

$$\bigwedge_{j\in B} (r_{\mathtt{add}_j} = \sum_{\substack{i\in\{S\cup B\}\\ Sub(j,i)\\ \mathtt{emit}_i}} r_{\mathtt{emit}_i})$$

$$\bigwedge_{j\in B} (r_{\mathtt{add}_j} > 0 \iff \mathtt{add}_j)$$

$$\bigwedge_{j\in B} (\mathtt{process}_j \Rightarrow r_{\mathtt{process}_j} > 0)$$

$$\bigwedge_{j\in B} (r_{\mathtt{process}_j} > 0 \Rightarrow ((\mathtt{process}_j \land \mathrm{X} r_{\mathtt{process}_j} = r_{\mathtt{process}_j})\,\mathbf{U}\,(\mathtt{emit}_j \lor \mathtt{fail}_j) \land (r_{\mathtt{process}_j} > 0)\mathbf{S}(\mathtt{take}_j \lor \mathtt{orig}))$$

$$\bigwedge_{j\in B} (r_{\mathtt{process}_j} = 0 \iff (\mathtt{orig} \lor (\neg\mathtt{process}_j \land \mathbf{Y}(\neg\mathtt{process}_j \lor \mathtt{emit}_j))))$$

$$\bigwedge_{\substack{j\in B\\ final(j)}} (r_{\mathtt{emit}_j} = 0)$$

$$\bigwedge_{\substack{j\in B\\ \neg final(j)}} \left(\mathtt{emit}_j \Rightarrow \begin{pmatrix} (\mathtt{buffer}_j = Y\mathtt{buffer}_j + r_{\mathtt{process}_j}) \\ \land(r_{\mathtt{emit}_j} \leq \sigma_j \mathtt{buffer}_j + d_j) \\ \land(r_{\mathtt{emit}_j} > \sigma_j \mathtt{buffer}_j + d_j - 1) \end{pmatrix} \land \begin{pmatrix} (X\mathtt{buffer}_j \geq \mathtt{buffer}_j - \frac{r_{\mathtt{emit}_j}}{\sigma}) \\ (X\mathtt{buffer}_j < \mathtt{buffer}_j - \frac{r_{\mathtt{emit}_j}}{\sigma} + 1) \end{pmatrix}\right)$$

$$\bigwedge_{\substack{j\in B\\ \neg final(j)}} (\neg\mathtt{emit}_j \Rightarrow (r_{\mathtt{emit}_j} = 0 \land (X\mathtt{buffer}_j = \mathtt{buffer}_j)\mathbf{UX}(\mathtt{emit}_j)))$$

Each bolt has a maximum take rate $\hat{r}_{\mathtt{take}_j}$ that limits the number of tuples that can be taken from the receive queue at any moment. If the elements in the queue are greater than or equal to $\hat{r}_{\mathtt{take}_j}$ and a *take* is performed by the bolt, then exactly $\hat{r}_{\mathtt{take}_j}$ tuples are taken from the queue. Otherwise, all the tuples are taken.

$$\bigwedge_{j\in B} ((\mathtt{take}_j \land \hat{r}_{\mathtt{take}_j} \geq q_j + r_{\mathtt{add}_j}) \Rightarrow (r_{\mathtt{process}_j} = q_j + r_{\mathtt{add}_j}))$$

$$\bigwedge_{j\in B} ((\mathtt{take}_j \land \hat{r}_{\mathtt{take}_j} < q_j + r_{\mathtt{add}_j}) \Rightarrow (r_{\mathtt{process}_j} = \hat{r}_{\mathtt{take}_j}))$$

Since the spouts are modeled as node which only emit tuples into the topology, we only define the emitting rate $r_{\mathtt{emit}_j}$ that is composed of the "nominal" emitting rate $\overline{r}_{\mathtt{emit}_j}$ (i.e. the emitting rate that the spout would if no failure ever happened) and the additional rate $r_{\mathtt{failure}_j}$ expressing how many tuples have to be re-emitted by spout j due to failures in the topology. Similarly to the bolt, each spout can emit up to a maximum quantity of tuple every time, excluding the replay tuples. Such maximum quantity is expressed by $\hat{r}_{\mathtt{emit}_j}$.

$$\bigwedge_{j\in S} ((\overline{r}_{\mathtt{emit}_j} \geq 0) \land (\overline{r}_{\mathtt{emit}_j} \leq \hat{r}_{\mathtt{emit}_j}) \land (r_{\mathtt{replay}_j} \geq 0))$$

$$\bigwedge_{j\in S} (\mathtt{emit}_j \Rightarrow (\overline{r}_{\mathtt{emit}_j} > 0) \land (r_{\mathtt{emit}_j} = \overline{r}_{\mathtt{emit}_j} + r_{\mathtt{replay}_j}))$$

$$\bigwedge_{j\in S} (\neg\mathtt{emit}_j \Rightarrow (r_{\mathtt{emit}_j} = 0))$$

Failure Propagation

In our model, whenever a node fails, the tuples being processed by the the node, together with the tuples in its receive queue, are considered as failed (not fully processed by the topology). According to the reliable implementation of Storm, the spout tuples that generated them must be resubmitted to the topology.
Since we do not keep track of the single tuples, but we only consider quantities of tuples throughout the topology, given an arbitrary amount of failed tuples, we can estimate the amount of spout tuples that have to be re-emitted by the connected spouts.

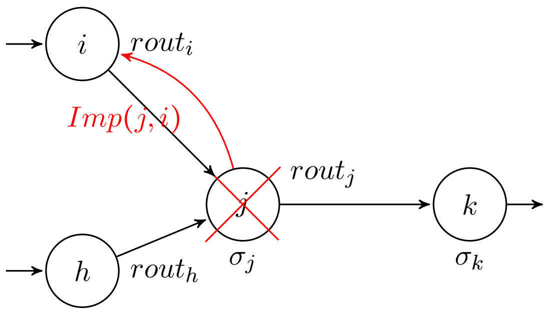
In order to express this relationship between the failing tuples in a specific (failing) node and the new tuples having to be re-emitted, we introduce the concept of *impact* of the node failure with respect to another (connected) node.

Such impact can be precomputed given the topology and we define it as follows:

$Imp(j,i)$ ("impact of node j failure on node i") is the coefficient expressing the ratio $\frac{tuples\_to\_be\_replayed(i)}{failed\_tuples(j)}$ where $j \in B$ is the failing bolt and $i \in \{S \cup B\}$ is another node in the topology.
If exist a path $path(j,i) = \{p_0, \dots, p_n | n > 0, p_0 = j, p_n = i\}$ connecting the two nodes such that $\forall k \in [0, n-1] Sub(p_k, p_{k+1})$, then a failure of node j has an impact on node i and $Imp(j,i) > 0$. If such a path does not exist, $Imp(j,i) = 0$.
In order to define how to calculate $Imp(j,i)$ over a generic path we first show how to obtain its value for two basic cases:

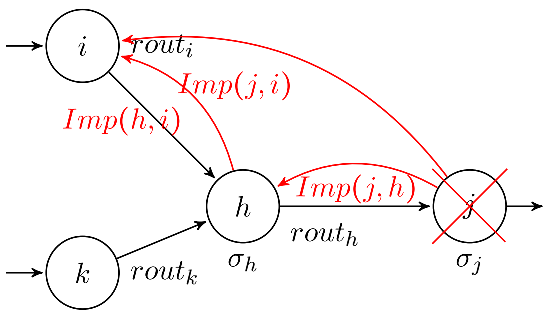
- if two nodes j and i are directly connected:

$$Sub(j,i) \Rightarrow (Imp(j,i) = \frac{r_{out_i}}{\sum_{\substack{k \in \{S \cup B\} \\ Sub(j,k)}} r_{out_k}})$$



- if the nodes j and i are connected by path passing through another node h:

$$(Sub(j,h) \wedge Sub(h,i)) \Rightarrow \left(Imp(j,i) = Imp(j,h) \cdot \frac{1}{\sigma_h} \cdot Imp(h,i)\right)$$



In general, if there is a path $path(j,i) = \{p_0, \dots, p_n | n > 1, p_0 = j, p_n = i\}$ defined as above:

$$Imp(j,i) = Imp(p_0, p_1) \cdot \prod_{k=1}^{n-1} \frac{1}{\sigma_k} \cdot Imp(k, k+1)$$

Once this coefficient is calculated for all the couples of $(bolt, spout)$ in the topology, it allows to determine the number of tuples to be re-emitted by each spout after a bolt failure by simply multiplying the number of failed tuples by the appropriate coefficient.

Failure Rates Behaviour - A (Single bolt formulae)

$r_{\texttt{replay}_j}$ is the quantity of tuples that need to be replayed due to a failure:

$$\bigwedge_{i \in S} \Big(r_{\texttt{replay}_i} = \sum_{\substack{j \in B \\ Imp(j,i)>0}} r_{\texttt{failure}_{ji}} \cdot Imp(j,i) \Big)$$

The impact of failure is therefore employed to calculate $r_{\texttt{failure}_{ji}}$ for each failing bolt. The behaviour of $r_{\texttt{failure}_{ji}}$ is defined as follows:

$$\bigwedge_{\substack{i \in S \\ j \in B \\ Imp(j,i)>0}} ((\texttt{startFailure}_j \wedge \neg\texttt{emit}_i) \Rightarrow (\mathrm{X} r_{\texttt{failure}_{ji}} = r_{\texttt{failure}_{ji}} + (q_j + r_{\texttt{process}_j} + r_{\texttt{add}_j}))))$$

$$\bigwedge_{\substack{i \in S \\ j \in B \\ Imp(j,i)>0}} ((\texttt{startFailure}_j \wedge \texttt{emit}_i) \Rightarrow (\mathrm{X} r_{\texttt{failure}_{ji}} = q_j + r_{\texttt{process}_j} + r_{\texttt{add}_j}))$$

$$\bigwedge_{\substack{i \in S \\ j \in B \\ Imp(j,i)>0}} ((\neg\texttt{startFailure}_j \wedge \texttt{emit}_i) \Rightarrow (\mathrm{X} r_{\texttt{failure}_{ji}} = 0))$$

$$\bigwedge_{\substack{i \in S \\ j \in B \\ Imp(j,i)>0}} ((\neg\texttt{startFailure}_j \wedge \neg\texttt{emit}_i) \Rightarrow (\mathrm{X} r_{\texttt{failure}_{ji}} = r_{\texttt{failure}_{ji}}))$$

Clocks Formulae

In order to represent the duration of the various processing phases of each bolt we introduce different clocks:

- $t^0_{\text{phase}_j}$ and $t^1_{\text{phase}_j}$ measure the duration of the $\texttt{process}_j$, $\texttt{idle}_j$ and $\texttt{fail}_j$ phases for each bolt j and the time elapsed between one $\texttt{emit}_j$ and the next one for each spout i.
- $\texttt{clock}_{to\texttt{fail}_j}$ measures the *time to failure*, i.e. the time elapsing between the end of a failure and the beginning of the next one for each bolt j.

$$\bigwedge_{j \in \{S \cup B\}} (t^0_{\text{phase}_j} = 0 \Rightarrow \mathbf{X}((t^1_{\text{phase}_j} = 0)\mathbf{R}(t^0_{\text{phase}_j} > 0)) \wedge (t^1_{\text{phase}_j} > 0)$$

$$\bigwedge_{j \in \{S \cup B\}} (t^1_{\text{phase}_j} = 0 \Rightarrow \mathbf{X}((t^0_{\text{phase}_j} = 0)\mathbf{R}(t^1_{\text{phase}_j} > 0)))$$

Each clock $t^0_{\text{phase}_j}$ will be initially set to 0. We will use the shortand $t_{\text{phase}_j} \sim c$ to indicate the formula:

$$\begin{aligned} &(t^0_{\text{phase}_j} > 0 \wedge (t^1_{\text{phase}_j} \vee (t^1_{\text{phase}_j} > 0)\mathbf{S}(t^0_{\text{phase}_j})) \Rightarrow t^0_{\text{phase}_j} \sim c) \\ &\wedge \\ &(t^1_{\text{phase}_j} > 0 \wedge (t^0_{\text{phase}_j} \vee (t^0_{\text{phase}_j} > 0)\mathbf{S}(t^1_{\text{phase}_j})) \Rightarrow t^1_{\text{phase}_j} \sim c) \end{aligned}$$

Reset conditions:

- $t_{\mathtt{phase}_j}$ **for bolts** - start of processing, failure or idle phase.

$$\bigwedge_{j\in B}\Big((t_{\mathtt{phase}_j}=0) \iff (\mathtt{orig} \vee \mathtt{take}_j \vee (\mathtt{fail}_j \wedge \neg\mathbf{Y}(\mathtt{fail}_j)) \vee (\mathtt{idle}_j \wedge \neg\mathbf{Y}(\mathtt{idle}_j)))\Big)$$

- $t_{\mathtt{phase}_j}$ **for spouts** - clock resets every time the corresponding spout emits

$$\bigwedge_{i\in S}((t_{\mathtt{phase}_j}=0) \iff \mathtt{emit}_i)$$

- $\mathtt{clock}_{to\mathtt{fail}_j}$ **(bolts)** - time-to-failure clock resets every time a failure ends.

$$\bigwedge_{j\in B}((\mathtt{clock}_{to\mathtt{fail}_j}=0) \iff (\neg\mathtt{fail}_j \wedge \neg\mathbf{Y}(\neg\mathtt{fail}_j)))$$

Bolt processing duration

Single interval variant (currently used):

$$\bigwedge_{j\in B}\left(\begin{array}{l}(\mathtt{process}_j \Rightarrow \\ (\mathtt{process}_j \wedge \neg\mathtt{emit}_j)\mathbf{U}((t_{\mathtt{phase}_j} \geq \alpha_j - \epsilon) \wedge (t_{\mathtt{phase}_j} \leq \alpha_j + \epsilon) \wedge (\mathtt{fail}_j \vee \mathtt{emit}_j))\end{array}\right)$$

Multiple rate intervals variant:

$$\bigwedge_{r\in(0,\hat{r}_{\mathtt{take}_j}]}\left(\bigwedge_{j\in B}\left(\begin{array}{l}(\mathtt{process}_j \wedge r_{\mathtt{process}_j}=r) \Rightarrow \\ (\mathtt{process}_j \wedge \neg\mathtt{emit}_j)\mathbf{U} \\ ((t_{\mathtt{phase}_j} \geq \mathtt{proc\_times[r][0]}) \wedge (t_{\mathtt{phase}_j} < \mathtt{proc\_times[r][1]}) \wedge (\mathtt{fail}_j \vee \mathtt{emit}_j))\end{array}\right)\right)$$

Failure duration

$$\bigwedge_{j\in B}\left(\begin{array}{l}\mathtt{fail}_j \Rightarrow \\ (\mathtt{fail}_j\mathbf{U}((t_{\mathtt{phase}_j} \geq \mathtt{MIN\_REBOOT\_TIME_j}) \wedge (t_{\mathtt{phase}_j} < \mathtt{MAX\_REBOOT\_TIME_j}) \wedge \neg\mathtt{fail}_j)\end{array}\right)$$

Idle duration

$$\bigwedge_{j\in B}\left(\begin{array}{l}\mathtt{idle}_j \Rightarrow \\ (\mathtt{idle}_j\mathbf{U}((t_{\mathtt{phase}_j} \leq \mathtt{MAX\_IDLE\_TIME_j}) \wedge (\mathtt{take}_j \vee \mathtt{fail}_j)\end{array}\right)$$

Minimum time to failure

$$\bigwedge_{j\in B}\left(\begin{array}{l}\neg\mathtt{fail}_j \Rightarrow \\ (\neg\mathtt{fail}_j\mathbf{U}(\mathtt{clock}_{tofail_j} > \mathtt{MIN\_TTF_j}))\end{array}\right)$$

Spout emitting intervals

$$\bigwedge_{j\in\mathbf{S}}\left(\begin{array}{l}(\neg\mathtt{emit}_j\mathbf{S}\mathtt{orig}) \Rightarrow \\ (\neg\mathtt{emit}_j\mathbf{U}((t_{\mathtt{phase}_j} \le \mathtt{proc\_times[1][0]}) \wedge \mathtt{emit}_j))\end{array}\right)$$

$$\bigwedge_{r\in(0,\hat{r}_{\mathtt{emit}_j}]}\left(\bigwedge_{j\in\mathbf{S}}\left(\begin{array}{l}(\mathtt{emit}_j \wedge \bar{r}_{\mathtt{emit}_j} = r) \Rightarrow \\ (\mathbf{X}(\neg\mathtt{emit}_j\mathbf{U}((t_{\mathtt{phase}_j} \ge \mathtt{proc\_times[r][0]}) \wedge (t_{\mathtt{phase}_j} < \mathtt{proc\_times[r][1]}) \wedge \mathtt{emit}_j)))\end{array}\right)\right)$$

Possible properties to be verified

- The queue of the bolt j is greater than MAXSIZE

$$\mathbf{F}(q_j > \mathtt{MAXSIZE})$$

- $clock_{fail}$ of bolt i is always less than T and bolt i eventually fails and the queue of j is eventually empty.

$$\mathbf{G}(fail_i \iff clock_{fail,i} < \mathtt{T}) \wedge \mathbf{F}(fail_i \wedge q_j = 0)$$

- bolt i has empty queue but is not failed and the queue remains empty for more than T.

$$\begin{array}{c}\mathbf{G}((q_i = 0) \wedge \mathbf{Y}(q_i > 0) \iff clock = 0)) \\ \wedge \\ \mathbf{F}(q_i = 0 \wedge \neg fail \wedge (q_i = 0\,\mathbf{U}(q_i = 0 \wedge clock > \mathtt{T})))\end{array}$$

A.2 First Order Logic Model.

The formalization[18] consists of the following state variables: t represents real time, $flag$ forces that the transition σ_{22} is fired immediately after σ_{21}, $statechange$ allows the system either to change the state (if the value is set to 1), i.e., σ_1 is fired, or the time just elapses (if the value is set to 0), $S(i)$ represents a spout which is multiplied n times for each process i, $B(i)$ represents a bolt which is multiplied n times for each process i, $L(i)$ is the length of the bolt B in the process i, $P(i)$ contains the number of tuples that were processed by the process i in the bolt $B(i)$ since last $Ta(K)e$ (by the transitions σ_3 and σ_4), $s_{time}(i)$ measures the time a spout emits in the process i (a spout emits at least T^{spout}_{min} time units and at most T^{spout}_{max} time units; after T^{spout}_{max} another process can operate on the spout), $bEmitTakeTime(i)$ represents the time elapsed since $P(i) = 0$ (which can happen when the bolt $B(i)$ is in the $(E)mit$ or $E(X)ecute$ state, in $\sigma_3, \sigma_4, \sigma_5$) and the bolt $B(i)$ is $(E)mit$, $wasBEmitting$ verifies if a certain spout was in the $(E)mit$ state in the past, $wasBTaking$ verifies if a a bolt was in the $Ta(K)e$ state,

[18]For the formalization we used both Cubicle and MCMT. In Cubicle, the model has approximatively the same number of transitions as the model described informally here, however in MCMT the model is visibly larger in the number of transitions: case distinctions have to be written explicitly since the tool does not do this automatically

$canTimeElapse$ was introduced such that σ_5 is not fired successively (constant c can be taken as big as necessary so many σ_5 transitions can be composed).

The $Init$ state of the system is described by the following formula.

$$\begin{array}{ll} t, flag, statechange = 0 & \wedge \\ \underset{i}{\forall}\,(S(i), B(i) = I,\ \ L(i), P(i), s_{time}(i), bEmitTakeTime(i) = 0) & \wedge \\ wasBEmitting, WasBTaking = 0\ \wedge\ canTimeElapse = 1 & \end{array}$$

The set of transitions is described bellow. If a state variable is not mentioned in a transition then it is assumed to be unchanged.

$$\begin{array}{ll} t, flag, statechange = 0 & \wedge \\ \underset{i}{\forall}\,(S(i), B(i) = I,\ \ L(i), P(i), s_{time}(i), bEmitTakeTime(i) = 0) & \wedge \\ wasBEmitting, WasBTaking = 0\ \wedge\ canTimeElapse = 1 & \end{array}$$

The set of transitions is described bellow. If a state variable is not mentioned in a transition then it is assumed to be unchanged.

$$\sigma_{1a}:\ \underset{x,y}{\exists}\ statechange = 1 \wedge flag = 0 \wedge$$
$$\underset{j}{\forall}\left(\begin{array}{lll} statechange' = & 0 & \wedge \\ S'(j) = & \text{if } j = x \text{ then } (E \text{ or } I) \text{ else } S(j) & \wedge \\ B'(j) = & \text{if } (j = y \text{ and } B(j) = E) \text{ then } (I \text{ or } K) \text{ else } B(j) & \\ & \text{elseif } (j = y \text{ and } B(j) = I) \text{ then } K \text{ else } B(j) & \wedge \\ canTimeElapse' = & 1 & \end{array}\right)$$

$$\sigma_{1b}:\ \underset{x}{\exists}\ B(x) = X \wedge P(x) = 0 \wedge$$
$$\underset{j}{\forall}\left(\begin{array}{lll} statechange' = & 0 & \wedge \\ B'(j) = & B'(j) = \text{ if } j = x \text{ then } E \text{ else } B(j) & \wedge \\ wasBEmitting' = & 1 & \wedge \\ canTimeElapse' = & 1 & \end{array}\right)$$

$$\sigma_{21}:\ \underset{x,c}{\exists}\ c > 0\ \wedge\ S(x) = E \wedge flag = 0\ \wedge T_{min}^{spout} < s_{time}(x) + c < T_{max}^{spout}\ \wedge$$
$$\underset{j}{\forall}\left(\begin{array}{lll} t' = & t + c & \wedge \\ flag' = & 1 & \wedge \\ statechange' = & 1 & \wedge \\ P'(j) = & \text{if } B(j) = X \text{ and } P(j) - Execrate * c \geq 0 & \\ & \text{then } P(j) - Execrate * c \text{ else } 0 & \wedge \\ s'_{time}(j) = & \text{if } j = x \text{ then } 0 \text{ else } s_{time}(j) + c & \wedge \\ canTimeElapse' = & 1 & \end{array}\right)$$

$$\sigma_{22}:\ \underset{x,y,c}{\exists}\ c > 0 \wedge S(x) = E\ \wedge\ L(y) + c \leq Lenmax \wedge flag = 1 \wedge$$
$$\underset{j}{\forall}\left(\begin{array}{lll} flag' = & 0 & \wedge \\ statechange' = & 1 & \wedge \\ L'(j) = & \text{if } j = y \text{ then } L(j) + c \text{ else } L(j) & \wedge \\ canTimeElapse' = & 1 & \wedge \end{array}\right)$$

$$\sigma_3: \quad \exists_x B(x) = K \ \wedge \ L(x) \geq Takemax \ \wedge flag = 0 \wedge$$
$$\forall_j \left(\begin{array}{lll} statechange' = & 0 & \wedge \\ B'(j) = & \text{if } j = x \text{ then } X \text{ else } B(j) & \wedge \\ L'(j) = & \text{if } j = x \text{ then } L(j) - Takemax \text{ else } L(j) & \wedge \\ P'(j) = & \text{if } j = x \text{ then } Takemax \text{ else } P(j) & \wedge \\ bEmitTakeTime\prime(j) = & 0 & \wedge \\ wasBTaking' = & 1 & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_4: \quad \exists_x \ B(x) = K \ \wedge \ 0 < L(x) < Takemax \wedge flag = 0 \wedge$$
$$\forall_j \left(\begin{array}{lll} statechange' = & 0 & \wedge \\ B'(j) = & \text{if } j = x \text{ then } X \text{ else } B(j) & \wedge \\ L'(j) = & \text{if } j = x \text{ then } 0 \text{ else } L(j) & \wedge \\ P'(j) = & \text{if } j = x \text{ then } L(j) \text{ else } P(j) & \wedge \\ bEmitTakeTime\prime(j) = & 0 & \wedge \\ wasBTaking' = & 1 & \wedge \\ canTimeElapse' = & 1 & \end{array} \right)$$

$$\sigma_5: \quad \exists_c \ c > 0 \wedge canTimeElapse = 1 \ \wedge \ flag = 0 \wedge$$
$$\forall_j \left(\begin{array}{lll} t' = & t + c & \wedge \\ statechange' = & 1 & \wedge \\ P'(j) = & \text{if } (B(j) = X \ \wedge \ P(j) - Execrate * c \geq 0) & \\ & \text{then } P(j) - Execrate * c & \\ & \text{elseif } B(j) \neq X \text{ then } P(j) \text{ else } 0 & \wedge \\ s'_{time}(j) = & s_{time}(j) + c & \wedge \\ bEmitTakeTime\prime(j) = & bEmitTakeTime(j) + c & \wedge \\ canTimeElapse' = & 0 & \end{array} \right)$$