A Model-Driven Approach for the Formal Verification of Storm-Based Streaming Applications

Francesco Marconi
Politecnico di Milano
Milan, Italy
francesco.marconi@polimi.it

Marcello M. Bersani
Politecnico di Milano
Milan, Italy
marcellomaria.bersani@polimi.it

Matteo Rossi
Politecnico di Milano
Milan, Italy
matteo.rossi@polimi.it

ABSTRACT

Data-intensive applications (DIAs) based on so-called Big Data technologies are nowadays a common solution adopted by IT companies to face their growing computational needs. The need for highly reliable applications able to handle huge amounts of data and the availability of infrastructures for distributed computing rapidly led industries to develop frameworks for streaming and big-data processing, like Apache Storm and Spark. The definition of methodologies and principles for good software design is, therefore, fundamental to support the development of DIAs. This paper presents an approach for non-functional analysis of DIAs through D-VerT, a tool for the architectural assessment of Storm applications. The verification is based on a translation of Storm topologies into the CLTLoc metric temporal logic. It allows the designer of a Storm application to check for the existence of components that cannot process their workload in a timely manner, typically due to an incorrect design of the topology.

CCS Concepts

•**Theory of computation** → *Verification by model checking;* •**Software and its engineering** → *Model-driven software engineering;*

Keywords

Formal Verification; Apache Storm; MDE; Data-intensive Applications; Temporal Logic

1. INTRODUCTION

Data-intensive applications (DIAs) are computational systems that process, in a relative small amount of time, huge amounts of diversified information usually produced by data sources with high throughput. Some of the most popular companies nowadays—e. g., Twitter (www.twitter.com), Groupon (www.groupon.com), Spotify (www.spotify.com), etc.—make large use of DIAs to process data gathered from millions of users.

http://dx.doi.org/10.1145/3019612.3019769

DIAs constitute a significant asset for the production of large-scale software, and have been drawing the attention of both academia and industry. The creation of frameworks that support designers over the entire life-cycle (design, development, testing, deployment, maintenance) of DIAs is of crucial importance, and constitutes a key research challenge in this area. Topics such as techniques and tools for quality assessment, architecture enhancement, agile delivery and continuous testing of DIAs are targeted by ongoing research projects like, for instance, the DICE European project [8].

The design approach envisioned by DICE is founded on model-driven principles and can be summarized as follows. The design of an application is decomposed into three distinct and consecutive phases, each one associated with a profiled UML diagram. Each phase focuses on a specific aspect of the design and represents a refinement of the previous one that has to be validated before starting the new refinement step. If design flaws are detected, designers can either change the current model, or modify the one built in the previous step, then redo the refinement. The design process starts from a conceptual model of the application, called Platform-Independent Model (PIM); this is refined, in the second step, into the so-called Platform-Specific Model (PSM), which provides the architectural schema of the application based on a specific (data-intensive) technology; finally, in the last step, the architectural model is refined to obtain a deployment model.

Nowadays, the frameworks promoting the development of DIA can be considered mature technologies. This fact is witnessed by the spread and the popularity of streaming and data-mining industrial applications in the IT market. After decades of research and industrial development, however, most of the frameworks lack tools for the analysis of the applications at design time. Nonetheless, they commonly are equipped with monitoring platforms that allow designers to manually inspect the running applications by means of statistics based on metrics measuring the processing time, the latency of the application, the throughput of the nodes and so on. We approach the assessment of DIAs by applying formal verification to the architectural models described through (metric) temporal logic. The goal of the analysis is to determine, at *design time* and through *automated techniques*, whether the behavior entailed by the architecture of the application conforms to specific properties over time. The properties that an application should satisfy typically depend on the technology adopted to implement the appli-

cation. For instance, we employed a logic-based modeling technique for the analysis of DIA in [14] and in [13].

Most of the available data-intensive frameworks allow designers to specify the architecture of DIAs as a directed graph whose nodes are computational resources which carry out specific operations. The semantics underlying a graph, which reflects the runtime behavior of the application, is determined by the target technology (e. g., the same graph has two different interpretations in case we adopt a streaming or a batch technology). In this paper, we consider Apache Storm [1], a popular technology for stream-based applications. The architecture of a Storm application is defined by means of a *topology*—i. e., a directed graph—where nodes are of two kinds: *computational nodes*, which implement the logic of the application by elaborating information and producing an outcome; and *input nodes*, which bring information into the application from its environment.

Various are the resources on the web that point out criteria guiding the design of Storm topologies, such as [4]. In most of the cases, the designers can follow guidelines [16, 4] that facilitate the analysis of the application by using the statistics that are available from the monitoring platform. To the best of the authors' knowledge, however, there are no formal techniques for the analysis of temporal properties at design-time.

This paper presents a different perspective. First, the model-driven approach fostered by the DICE workflow is conducted by means of a simple application, which is developed through an iterative refinement process. The complete verification workflow of the architectural model is carried out according to the concepts included in the DICE UML profile [10] and the validation of the topology is done through the analysis performed by D-VerT [2], the DICE tool that allows the verification of Storm topologies at design-time. We also focus on all the necessary transformations needed for translating the UML diagram of the architecture of the DIA into a formula of the CLTLoc metric temporal logic [7], which is solved by D-VerT to validate the Storm architecture represented by the UML model. Finally, we presents the verification tool, which is the component implementing the transformations. Furthermore, we introduce an industrial use case that is provided by one of the partner in the DICE consortium and we use it to validate our verification approach. We set up an experiment to compare the result obtained with D-VerT and the behavior of the application at runtime. The real application has been abstracted by means of a topology characterized by the same non-functional properties, which has been implemented using stub components that mimic the behavior of the nodes of the real application. Then, through the monitoring platform, the topology has been analyzed and the resulting behavior has been compared with the D-VerT outcome.

The paper is structured as follows: Section 2 presents some background notions on Apache Storm and briefly recaps our approach to the modeling of Storm topologies with temporal logic introduced in [14]. Section 3 introduces the methodology for the verification of Storm topologies based on formal validation of refined UML models. Section 4 describes the structure of D-VerT and the transformations needed for enabling the verification of architectural models. Section 5

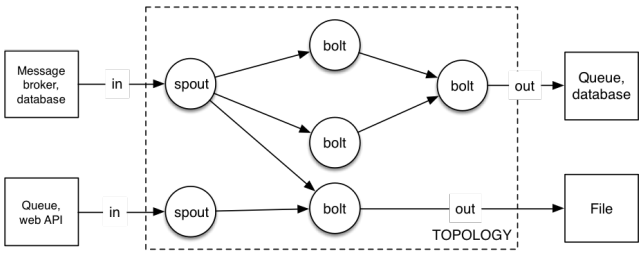

Figure 1: Example of Storm topology.

shows the application of the methodology through and example of Storm application which, at the end, undergoes verification with D-VerT. Section 6 presents another use case for the tool and addresses the validation of the verification results by monitoring the topology. Section 7 briefly discusses some related works, and Sect. 8 concludes.

2. BACKGROUND

2.1 Apache Storm

Apache Storm [1] is a stream processing system, developed and open sourced by Twitter in 2012, allowing real-time processing of large-scale streaming data on horizontally scalable systems through a parallel and distributed computation.

The computational model of Storm applications is the Storm topology, i. e., a directed graph whose nodes realize the operations performed over the data flowing through the application and whose edges indicate how such operations are combined. Data is encoded into streams that are infinite sequences of structured messages, also named tuples, which are processed by the application.

A topology node set consists of *spouts* and *bolts* (in the following also referred to as *topology components*). Spouts are stream sources which usually get data from external systems, such as queuing brokers (e. g., Kafka, RabbitMQ, Kestrel), or from other data sources (e. g., Twitter Streaming APIs), whereas bolts transform the incoming data streams into new output streams to be processed by the connected bolts. Connections are statically defined at design time by the subscription of the bolts to other spouts or bolts. Fig. 1 shows an example of Storm topology that will be used in Sect. 5.

Storm is capable of guaranteeing the so-called "at least once" message processing. Reliable spouts keep track of all the tuples they emit, and if one of them fails to be processed by the entire topology within a certain timeout, then the spout re-emits it into the topology. When message processing is "best-effort", instead, (unreliable) spouts emit each tuple only once, without checking for the successful completion of the processing. Bolts usually perform operations, such as filtering, join, functions, database interaction, which are combined through the topology to perform complex transformations. The Java interfaces defining a bolt include the `execute()` method that implements its functionality; it reads the input tuples, processes the data, and emits (via the `emit()` method) the transformed tuples on the output streams. When the spouts are reliable, bolts have to acknowledge the successful or failed processing of each tuple at the end of the

execution.

Storm is designed to be executed on distributed clusters and leverage their computational power. A deployed topology is composed of one *master node* and several *worker nodes*. Each worker node instantiates one or more *worker processes* to enable the execution of the functionalities implemented by spouts and bolts belonging to the same topology. Each worker process runs a JVM and spawns therein one or more *executors* (i.e. threads). Executors run one or more *tasks* which, in turn, execute either a spout or a bolt.

Running a topology requires the definition of a number of parameters, among which:

- the number of executors running in parallel each component, either spout or bolt (i.e., the *parallelism* associated with the execution of the component) and
- the total number of tasks over those executors.

Since each executor corresponds to a single thread, multiple tasks run serially on the same executor, though the default option is one task per executor.

Communication among workers and executors is managed through a multi-level queuing system. Each executor has its own input queue and output queue. Tuples are read from the input queue by the thread handling the spout/bolt logic; afterwards, when they are emitted, they are put into the output queue and later moved to the worker's transfer queue by means of a send thread running within the executor. Every worker runs a *Receive thread* and a *Send thread*. The former listens for incoming tuples appended in a *worker's receive queue* and, based on the received message and the nodes configuration, forwards them to the appropriate executors' input queue; the latter, instead, gets the tuples from the executors' output queue, puts them in a *worker's transfer queue*, and forwards them either to other workers or to other executors in the same worker. Fig. 2 depicts the internal structure of a worker queueing system and shows the executor input and output queues.

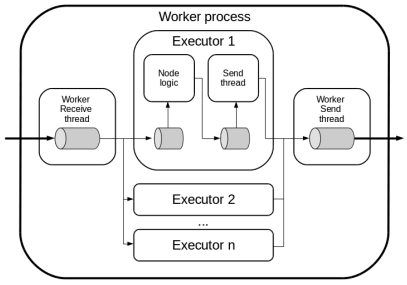

Figure 2: Queueing system of a worker process.

A common issue of distributed applications is caused by bottleneck nodes. In Storm applications, this criticality can take place when some executors of the topology work "at capacity". This term refers to a metric of great interest for software engineers as the designers consider the *capacity* of bolts to evaluate the topology performance before the deployment of the application. The *capacity* of a bolt [1] is calculated by the Storm monitoring service and it is defined as the percentage of the time that the bolt expends in processing the incoming tuples, with respect to a given time interval where the estimation is calculated. This value can be calculated for every bolt of the topology by means of Formula (1), using the following metric values:

$n_{executed}$: it is the total number of incoming tuples that have been processed by a bolt in the last T millisecs.

latency : it is the average time that a tuple spent in the `execute`() method of the bolt in the last T millisecs.

$T_{capacity}$: total time to consider for estimating the capacity of the bolt, that is by default set to 10 minutes.

The capacity of a bolt is then defined as:

$$\frac{n_{executed} \cdot latency}{T_{capacity}} \tag{1}$$

A bolt with capacity close to 1 spends most of the time in processing the input, being characterized by a very small idleness. As remarked in various technical on-line resources about Storm performance, high capacity values should be prevented by adopting countermeasures that diminish the values when they are close to 1. In such a situation, a topology might fail the processing of the incoming data as the receive queue of the executors, running a node working "at capacity", might grow in size and reach the maximum limit. Therefore, new tuples that cannot be appended in the queue are irremediably lost, unless the expensive reliable processing is activated. The common practice to address this issue is to increase the parallelism of the bolts (i.e., the number of executors assigned to it) or the number of tasks that are instantiated in their executors. More refined actions can also be realized and usually employ accurate information that are collected by profiling the application [16].

2.2 Modeling Storm topologies

Our verification approach is founded on a temporal logic model that represents the computation of Storm topologies. In particular, the model is designed to discover unwanted behaviors of running topologies. To this end, it is specifically devised to be representative of the application runtime.

The definition of the model required first the identification of malfunctions of the application that software engineers designing Storm topologies consider to be critical at runtime. We identified some of the aspects that might cause such malfunctions and we focused on one of them, which is related to the bottleneck analysis of nodes mentioned in Sect. 2.1. Therefore, we defined our model in order to allow the analysis of the existence of bolts that would saturate their capacity as they cannot process the incoming stream of tuples on time, thus causing a monotonic growth of the size of their queues. The model of a topology captures how the timing parameters of its components—such as the delays between two consecutive spout events inputting tuples in the topology and the processing time of tuples for each bolt—affect the size of the bolts' queue.

The relevant aspects of the computation, such as the functionality that is implemented in every method `execute()`, are reflected in the modeling by means of an appropriate behavioral abstraction. For instance, emitting a tuple and storing it into a queue is modeled through an *emit* event that increments the size of the target queue. The behavior of the relevant features and parameters of spouts and bolts is extracted by reverse-engineering the Java interfaces of the Storm API.

Furthermore, some suitable assumptions are considered to generate models that can be practically managed by state-of-the-art formal verification tools in a reasonable amount of time.

- Deployment details, such as the number of worker nodes and the features of the (possibly) underlying cluster are abstracted away; topologies are assumed to run on a single worker process and each executor runs a single task, which is the default Storm configuration of the runtime.
- Each bolt has a single receive queue for all its parallel instances and no sending queue, while the workers' queues are not represented (single-worker scenario). For generality, all queues have unbounded size and the current number of tuples in a queue is represented by means of a positive integer value.
- The contents of tuples is not modeled and, since tuples are all assumed to have the same size, the size of queues is represented by the number of tuples they contain.
- The external sources of information abstracted by the spouts are not represented, since they are outside of the perimeter of the application. So, their queues are not represented.
- For each component, the duration of each operation or the permanence in a given state has a minimum and a maximum time.

The computation of Storm topologies is captured through a set of formulae written in the CLTLoc metric temporal logic [7] augmented with counters, which are used to represent the size of bolts' queues during the computation. CLTLoc [7] is a decidable extension of LTL where formulae are defined over a finite set of atomic propositions and a set of dense variables over $\mathbb{R}_{\geq 0}$ representing clocks. For instance, a possible atomic formula over clock x is $x < 4$. Similarly to TA, a clock x measures the time elapsed since the last reset, that occurs when $x = 0$. The interpretation of *clocks* is defined through a clock valuation assigning, for every time position $\mathbb{N}$, a real value to each clock x in the formula. The semantics of time adopted is *strict*, namely the value of a clock must strictly increase in two adjacent time positions, unless it is reset.

The complete description of the formal model of a topology can be found in [5]. It consists of four parts, which represent: (i) the evolution of the state of the nodes; (ii) the behavior of the queues; (iii) timing constraints; (iv) failures. [14], instead, presents some of the technical details of the adopted decision procedure, which is validated through some experimental results.

The behavior of a bolt is the most relevant part of the topology model. A bolt can alternatively be in one of the following states: *process*, *idle* or *failure*. If a bolt is idle and its queue is not empty, then it can start processing the tuples stored therein. The Storm supervisor is responsible for the activation of the threads running the method `execute()` on a selected tuple. In our model, this is represented through an instantaneous *take* action that models the action of removing tuples from the bolt's queue and marks the beginning of the processing. The execution of method `execute()` is represented by the state *execute*, which is part of macro-state *process*, together with the actions *take*, that delimits the beginning of the processing phase, and *emit* that occurs at the end of it. In our model, we indicate the latency of a bolt with α. Once the execution is completed, the bolt emits output tuples with an instantaneous action corresponding to the *emit* state. Bolts may fail and failures may occur at any moment; upon a bolt failure, the system goes to the *fail* state. If no failure occurs, after an *emit* a bolt goes to *idle*, where it stays until it reads new tuples.

To give a flavor of the formal model underlying our verification approach, we introduce a few examples of CLTLoc formulae. Formulae (2)-(3) capture how the number of elements in the queue of bolt j ($\mathtt{q}_j$) is updated whenever tuples are enqueued ($\mathtt{add}_j$) or dequeued ($\mathtt{take}_j$). They use $\mathbb{N}$-valued discrete counters to represent the amounts of tuples exchanged in the topology. For instance, $\mathtt{q}_i$ is the size of queue of bolt j. Term $\mathrm{X}\mathtt{q}_i$ represents the value of $\mathtt{q}_i$ in the next position of time. Every time the component j emits ($\mathtt{emit}_j$ holds), the queues of all bolts subscribing to j—i. e., those which are targets of arcs outgoing from j in the topology—receive $\mathtt{r}_{\mathtt{emit}_j}$ tuples—i. e., the variables $\mathtt{q}_i$ representing the occupancy level of those queues are incremented by $\mathtt{r}_{\mathtt{emit}_j}$. When multiple components subscribed by a bolt emit tuples simultaneously, the increment on its queue is equal to the sum of all the tuples emitted (the value of $\mathtt{r}_{\mathtt{add}_j}$ in Formulae (2)-(3)). Dually, when $\mathtt{take}_j$ holds, the occupancy level $\mathtt{q}_j$ is decremented by $\mathtt{r}_{\mathtt{process}_j}$ (number of tuples read by bolt j). Proposition $\mathtt{add}_j$ is true when at least one of the components subscribed by j is emitting, whereas $\mathtt{startFail}_j$ is true in the first instant of a failure state.

$$\mathtt{add}_j \wedge \neg\mathtt{take}_j \wedge \neg\mathtt{startFail}_j \Rightarrow (\mathrm{X}\mathtt{q}_j = \mathtt{q}_j + \mathtt{r}_{\mathtt{add}_j}) \quad (2)$$

$$\mathtt{take}_j \Rightarrow (\mathrm{X}\mathtt{q}_j = \mathtt{q}_j + \mathtt{r}_{\mathtt{add}_j} - \mathtt{r}_{\mathtt{process}_j}) \quad (3)$$

To measure the duration of each state and to impose timing constraints between events, we use a set of dense-time CLTLoc clock variables for each component of the topology. For example, Formula (4) imposes that when *emit* occurs, the duration of the current processing phase is between $\alpha - \epsilon$ and $\alpha + \epsilon$, where $\epsilon \ll \alpha$ is a positive constant that captures possible (small) variations in the duration of the processing.

$$\mathtt{process} \wedge \mathtt{emit} \Rightarrow (t_{\mathtt{phase}} \geq \alpha - \epsilon) \wedge (t_{\mathtt{phase}} \leq \alpha + \epsilon) \quad (4)$$

The formal model includes a number of parameters, such as α introduced above, capturing the features of the topology, which can be configured at design time. In addition to α, other parameters are, for bolts, a coefficient σ expressing the kind of operation performed by the bolt in terms of quantity of output tuples emitted given an input tuple, and also the minimum and maximum time to failure. Spouts, instead,

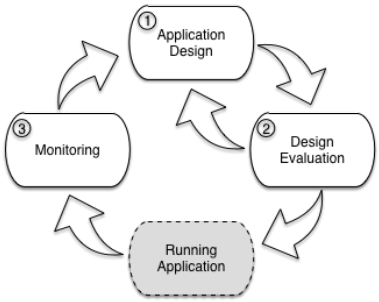

Figure 3: Iterative refinement process supported by the DICE framework.

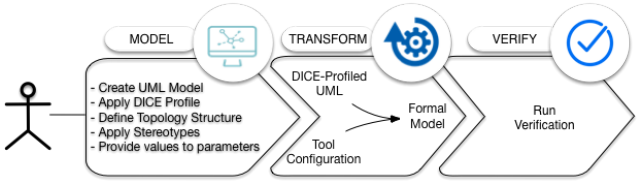

Figure 4: The main steps of D-VerT workflow.

are characterized by the average number of tuples emitted per time unit. Both spouts and bolts are also characterized by their level of parallelism, corresponding to the number of executors for the component.

3. ANALYSIS OF STORM TOPOLOGIES

D-VerT allows designers to validate temporal aspects of DIAs by means of a logic-based formal verification approach outlined in Section 2.2. The implementation of D-VerT currently supports the analysis of Storm topologies and Spark execution DAGs [6], but it can be easily extended to deal with other big-data technologies if and when their computational model is formalized through CLTLoc formulae.

D-VerT is part of a more complex design process which conforms to the principles of model-driven software engineering pursued by the DICE methodology. As illustrated in Fig. 3, the designer defines the application by means of domain-specific models with an iterative approach consisting of three steps: (i) application design, (ii) design evaluation and (iii) monitoring of a running deployed application. D-VerT is situated at the second level of the design workflow and enables the refinement of the architectural design of the application depending on the outcome of the formal analysis. The input of D-VerT is an annotated UML (class) diagram which specifies the architecture, i.e., the topology, of the application. In case of an incorrect timing design, the outcome of D-VerT consists of a possible execution of the topology causing an undesired accumulation of tuples in some bolts. In such a case, the designer can refine the application architecture by changing the values of some parameters of the UML diagram and then redo the evaluation until (possibly) all flaws affecting the model are removed. A different scenario, which also entails a design refinement, might occur when some parameter values that are measured on a running application differ from the values used for verification at design time. In such a situation, monitored data obtained from the deployed application can be exploited to update the model, which can then be newly verified.

Relying on UML profiles for modeling is a common practice in model-driven development as it allows for the creation of domain-specific languages by extending or restricting UML to fit a specific domain. A UML Profile is made of a set of stereotypes, tags, constraints and meta-models that allow the designer to represent artifacts in a specific domain. A stereotype is a meta-concept which is used to categorize an element in a model (for example, *container*) with specific notions belonging to a domain.

As shown in Fig. 4, at the starting point of the workflow the user creates an annotated UML model describing the relevant technological aspects of the architecture of a DIA. The UML model includes suitable design abstractions, capturing specific data-intensive technologies—Storm in our case—that are adopted for implementing the architecture of an application. The diagram, called DICE Technology Specific Model (DTSM), is at the PSM level (see Sect. 1) in the model-driven approach pursued by DICE. Specifically, in a DTSM diagram, a stereotype classifies an element of an application with aspects related to a specific technology. A DTSM diagram includes generic concepts that fit many data-intensive frameworks, such as `ComputationNode`, `StorageNode` or `SourceNode`, and specific ones, depending on the selected technology. In the case of Storm, the relevant features and aspects defining a Storm topology constitute the meta-model for designing Storm applications. Some of them, that are used in Sect. 4, are `Topology`, `Spout`, `Bolt` and `TopologyConfigurations`. For a comprehensive description of the concepts available in DTSM diagrams see [10].

DTSM diagrams for Storm include all the values of the parameters that are useful to carry out the analysis of a topology. As depicted in Fig. 4, verification of DTSM models is done through the automatic translation of the diagrams into a set of CLTLoc formulae, which are then analyzed by the Zot bounded satisfiability checker [3] using the technique presented in [14]. More precisely, Zot is fed the CLTLoc formulae capturing the application under design and the property to be checked concerning the unbounded growth of the queues of interest.

The tool produces one of two responses: (i) a trace of the modeled Storm topology—a *counterexample*—corresponding to an execution of the application in which one of the queues grows indefinitely—in this case, the set of formulae is *satisfiable* (SAT); or (ii) the notification that the set of formulae is *unsatisfiable* (UNSAT). In the latter case, since the language used to formalize Storm topologies is in general undecidable, we cannot conclude definitively that there is no execution of the application such that the queues grow indefinitely, but only that, within the bounds chosen for the search of counterexamples, none was found. Still, an UNSAT result increases our confidence that no major design flaws are present in the architecture of the Storm topology for what concerns its ability to process data in a timely manner.

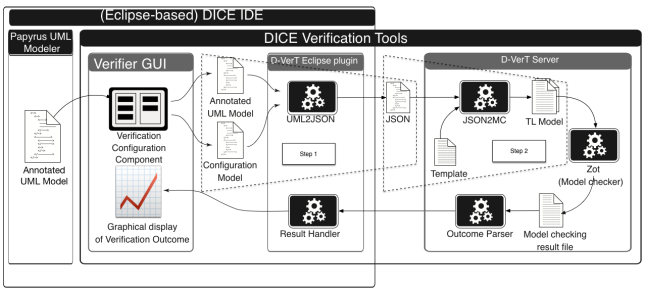

Figure 5: D-VerT workflow mapped onto the client-server architecture of the tool.

4. TOOL DESCRIPTION

This section outlines the architecture of the D-VerT tool, the transformation enabling the verification process and the kind of analysis currently supported by the tool.

4.1 Tool Architecture

As shown in Fig. 5, D-VerT is structured as a client-server application. The client component is an Eclipse plug-in, and is part of the DICE IDE. It allows users to define the design of the DIA under development, then, after providing some additional configuration information, to launch verification tasks and to retrieve their outcomes. The server component consists in a RESTful web application written in Python. The D-VerT server exposes APIs to launch verification tasks and to obtain information about their status. To simplify the setup and deployment steps, the D-VerT server is available as a Docker[1] container. The client-server architecture decouples the application design phase from the rather computationally-intensive verification phase. Based on the needs of the user, the D-VerT server can be instantiated either on the local machine or on a remote server.

4.2 Topology creation

The design of a DIA is specified through the DICE IDE, which is based on the Papyrus tool. As mentioned above, Storm topologies are described as DICE-profiled UML Class diagrams. Each computational node of the application is defined by introducing a class tagged with a stereotype specifying whether the node is a spout or a bolt. Depending on the stereotype applied, the designer defines the values for all the necessary parameters described in Sect. 2. Subscriptions (i. e., connections) of bolts to other components are modeled as associations between the corresponding classes.

4.3 Transformations

The verification process is made possible by a two-step transformation applied on the DICE-profiled UML models to obtain the corresponding set of CLTLoc formulae.

The first step of the transformation is performed in the D-VerT client by the UML2JSON module, which extracts from the DICE-profiled UML model all parameters that are relevant for the verification. These parameters are then serialized into a JSON object, which is used to invoke the server component. The extraction of the relevant features is done by suitably navigating the UML file. DIA components and their parameters are detected thanks to their being annotated with proper stereotypes from the DICE profile.

The second step takes place in the D-VerT server, which receives the request from the client, produces the corresponding formal model and feeds it to the underlying Zot [3] tool. More precisely, the JSON2MC module, based on the contents of the JSON object included in the request, generates the temporal logic model using a templating mechanism.

4.4 Analysis

In its current stage of development, D-VerT provides support for the analysis of the boundedness of bolts' queues. Through the run configuration dialog box of the tool (see Fig. 6) the designer can specify the bolts of interest, the depth of the search over which the verification has to be performed (the "time bound", which corresponds to the maximum length of the trace produced) and the Zot plug-in to be used. The analysis allows for the detection of possible runs of the system leading to an unbounded growth in the size of at least one of the aforementioned bolts' queues. This corresponds to the presence in the topology of at least one bolt that is not able to manage the incoming flow of messages in a timely manner. In this case the tool provides as output to the user the trace witnessing the existence of such a run of the system—i. e., the counterexample violating the boundedness property. The trace is returned to the user both in a textual format (i. e., the bare output of Zot) and in a graphical format, in order to provide a more user-friendly visual hint about the system execution. Figure 10 shows an example of such output trace, returned by the tool for the use case of Sect. 5. It represents the evolution of the number of tuples in the queue over time. The trace is composed by a prefix and a suffix: the latter, highlighted by the gray background, captures the growing trend of the queue size, as it corresponds to a series of operations in the system that can be repeated infinitely many times. When no trace is detected, the result is UNSAT.

5. D-VERT WORKFLOW IN ACTION

In this section we illustrate the usage flow of D-VerT for the iterative refinement of a Storm topology. The use case is taken from the open source project StormCrawler.[2] Suppose we want to create a web crawler application to efficiently fetch, parse and index web resources of our interest. Given the dynamic nature of the web, this kind of task can be formulated as a streaming problem, where the input consists in a continuous stream of URLs that need to be processed by the streaming application with low latency, and the output is represented by the resulting indexes.

We start by modeling the application at the PIM level. In this case, the model simply includes a *source node*, a *computation node* and a *storage node*, as depicted in Fig. 7. We decide to use a Kafka queue as source node, a Storm topology as computation node and ElasticSearch as storage node.

Since we are interested in analyzing the Storm topology,

[1] https://hub.docker.com/r/deibpolimi/d-vert-server

[2] https://github.com/DigitalPebble/storm-crawler

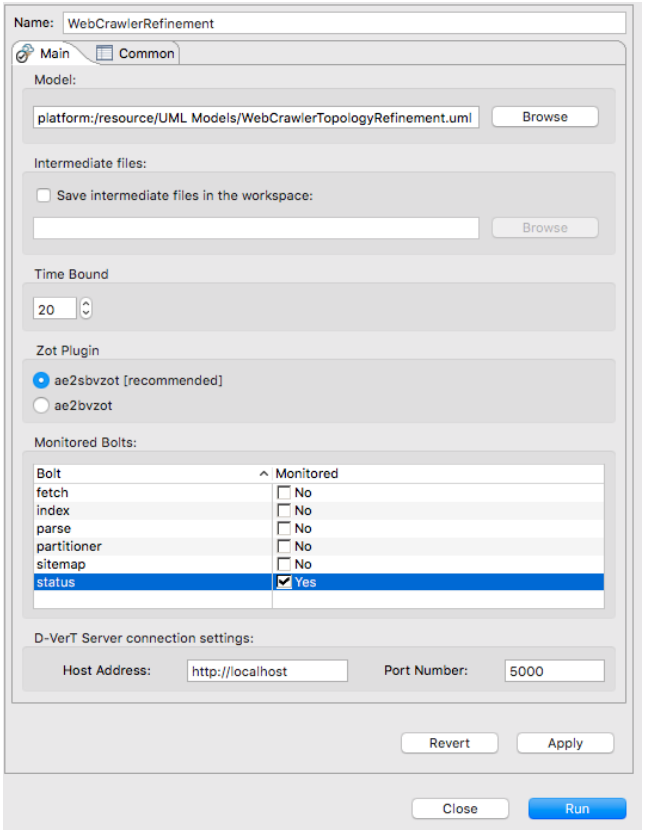

Figure 6: Run configuration view.

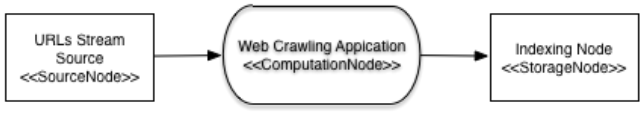

Figure 7: PIM of the web crawler application.

we focus on the computation node and consider the source node and the target storage node as "black boxes". At the PSM level we insert more technology-related aspects, such as, in the case of Storm, the topology structure and a series of non-functional characteristics. Figure 8 shows the PSM (DICE-profiled UML diagram) of the initial design of the topology. The configuration parameters are represented as UML comments for illustrative purposes. Notice that associations between components have the opposite direction with respect to the data flow among them, since they indicate the subscription of bolts to the associated components' streams. The diagram includes one spout in charge of fetching the input flow of URLs from Kafka and three bolts performing various steps of the web crawling process. The *partitioner* bolt partitions the incoming URLs, while the *crawler* bolt performs many operations such as resource fetching, metadata parsing and content indexing. The *status* bolt at the end of the chain indexes the URL, metadata and its status to a specific "status" Elasticsearch index. Each of these topology components can be executed by an arbitrary number of parallel threads, and is characterized by the

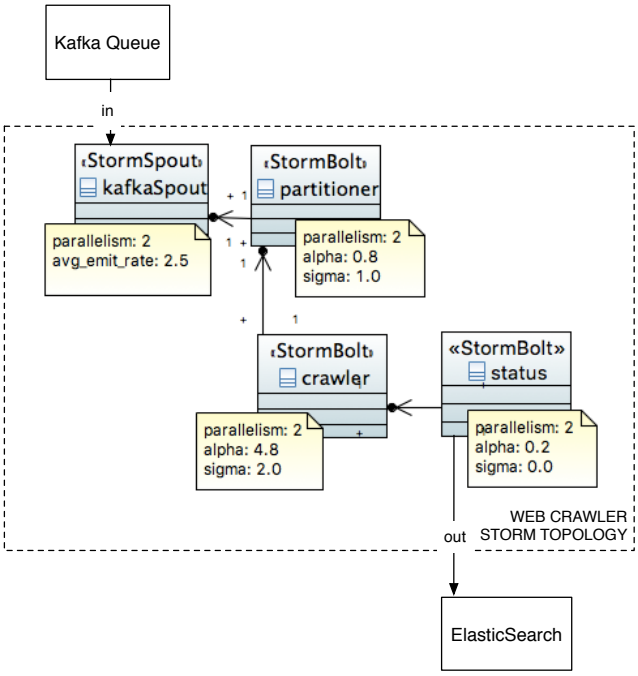

Figure 8: Initial PSM of the web crawler topology.

(average) execution time (time needed to perform its task) and by the (average) number of tuples emitted with respect to the number of tuples received as input. These aspects are specified as parameters in the UML class diagram. The formal analysis on the initial topology design helped us to detect an unbounded increase in the queue of the *crawler* bolt. This outcome from the tool led us to review the topology structure, and to decide for the decomposition of the *crawler* bolt in a series of bolts, each of them performing a subtask of the original bolt (*fetch*, *sitemap*, *parse* and *index*). The refined version of the topology, shown in Fig. 9, aims at lightening the load on the core crawling phase by pipelining the main operations and by directly updating the *status* bolt with partial results computed by the new bolts.

After the refactoring the tool revealed another unwanted run of the system, this time showing a growing trend in the queue of the *status* bolt (Fig. 10). This bolt, subscribing to the streams of the four newly-created bolts, needs a further refinement to avoid excessive loads in its input buffer. By increasing the parallelism level of the *status* bolt to 4, D-VerT was not able to detect any counterexample after tens of hours of execution, and returned an UNSAT result. Since, as already remarked, an UNSAT result would not be a guarantee of absence of counterexamples, in this work we focused on the detection of potential problems (SAT results) and their validation, as described in Sect. 6. Execution times for the verification vary significantly depending on the topology configuration, ranging from the 50 seconds of the first analysis (Fig. 8) to the many hours of the third analysis.[3]

[3]Experimental analysis carried out on commodity hardware (MacBook Air running MacOSX 10.11.4. with Intel i7 1.7 GHz, 8 GB 1600 MHz DDR3 RAM; SMT solver used by Zot was z3 v.4.5.0).

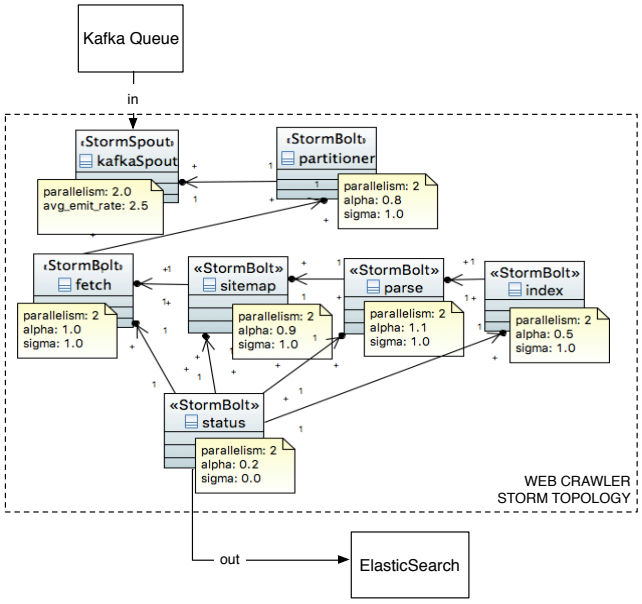

Figure 9: Refinement of the web crawler Storm topology PSM.

6. VALIDATION

In this section, we discuss the validation of our approach with an additional use case that will be analyzed by the D-VerT. The use case consists in a Storm application (*WebAnalysis* topology) which has been developed by an industrial partner in the DICE project. The application, similarly to the use case presented in Sect. 5, is designed to analyze a series of web resources in order to find, extract and categorize media items and articles. As shown in the PSM model of the topology in Fig. 11, the input data is fetched from a Redis[4] in-memory database by the *RedisSpout* component and, after a series of operations performed by the bolts of the topology, media and articles are then indexed on an instance of the open-source search platform Apache Solr[5] by the *SolrBolt* component.

[4]https://redis.io/

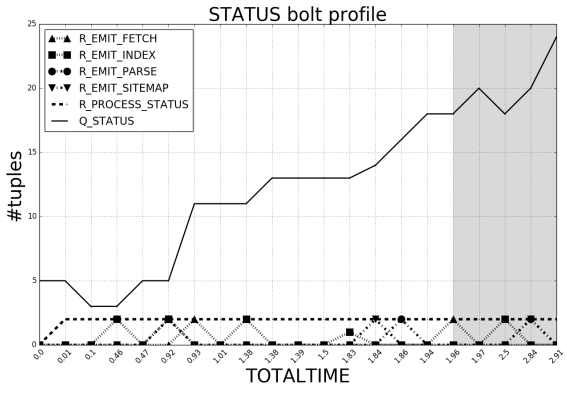

Figure 10: Graphical output trace of the *status* bolt returned by D-VerT. The black solid line represents the number of elements in the input buffer.

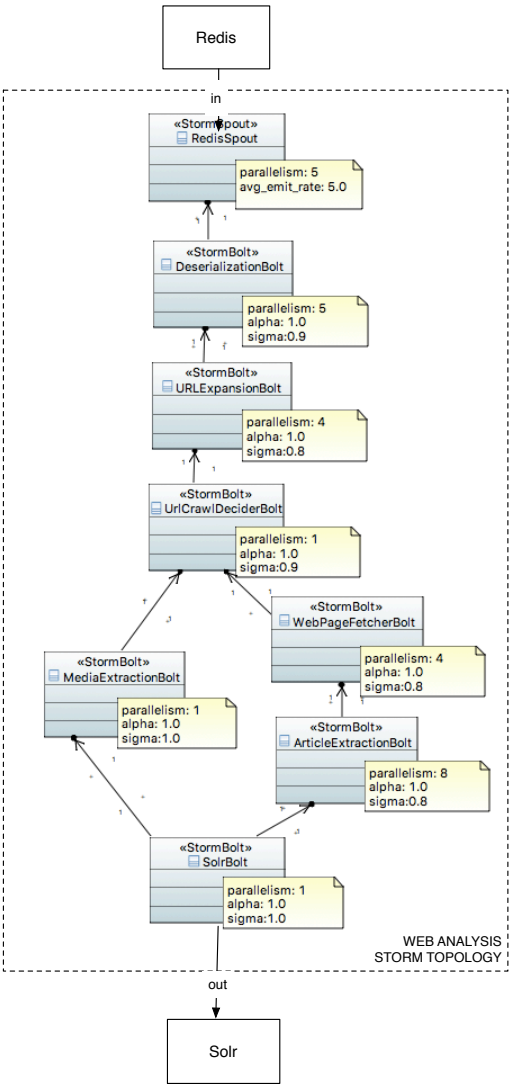

Figure 11: PSM model of the WebAnalysis topology

Our analysis spotted a potential problem in the input buffer of the *UrlCrawlDeciderBolt* component, as D-VerT returned a SAT result and provided an output trace showing an unbounded increase in the queue of the bolt (Fig. 12).

In order to validate the results, since the application code is closed-source, we have designed a topology with the same structure of the topology implementing the application and which was able to simulate the temporal characteristics of the *WebAnalysis* topology (in terms of α, σ, *parallelism*, etc.). We ran the application on commodity hardware and, during the execution, we collected all the relevant quality information provided by the Storm platform, such as the *capacity* of each bolt (introduced in Sect. 2.1), the *complete*

[5]http://lucene.apache.org/solr/

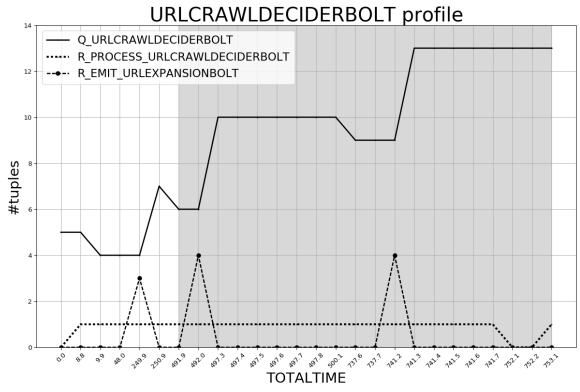

Figure 12: Graphical output trace of the UrlCrawlDeciderBolt returned by D-VerT

latency of the topology and the number of failed tuples due to timeout. To measure the complete latency of the processed tuples, we enabled the acking mechanisms available in Storm. Through this mechanism, a tuple is marked as failed after its complete latency exceeds the default threshold of 30 seconds. In other words, if a tuple has not been processed for more than 30 seconds then the tuple will be discarded by the topology because it is not valid anymore.

The monitored values showed that *capacity* of *UrlCrawlDeciderBolt* was constantly very close to 1, meaning that the bolt was almost always busy in processing tuples. This behavior usually implies that the bolt is not able to manage the incoming flow of data in a timely manner, negatively affecting the latency of the entire topology. The presence of a problem was confirmed by the evolution of the average *complete latency* measure, shown in Fig. 13 and Fig. 14. Each point shown in the graphs is the value of the average complete latency measured with respect to the last 10 minutes. The plot shows a steep increase in the first minutes of execution, followed by a sharp drop after the average complete latency goes beyond 20 seconds. The same fluctuating trend is repeated later in time. This behavior appears to be due to the timeout settings. When the average value of complete latency gets close to 20 seconds, most of the tuples accumulated in the topology are failed and discarded due to timeout. After this massive disposal, the decongestion of the input buffers allows the newly emitted tuples to be completely processed by the topology in less time. However, because of the bottleneck in the topology, the average complete latency keeps increasing and then follows the same trend.

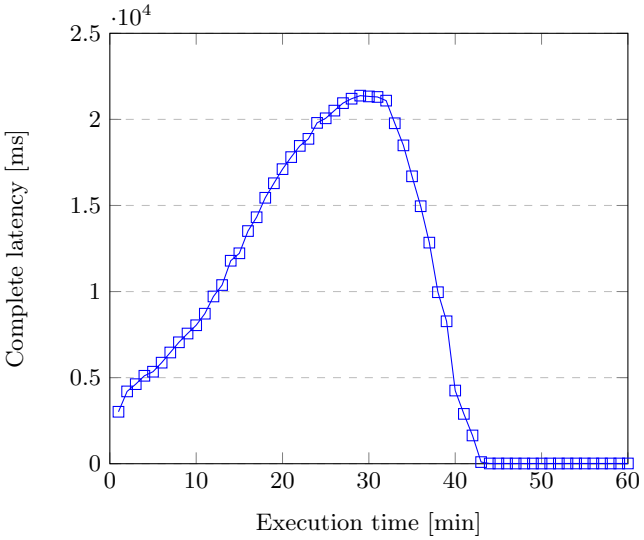

Figure 13: Average complete latency of the tuples emitted by RedisSpout of topology in Fig. 11 over one hour.

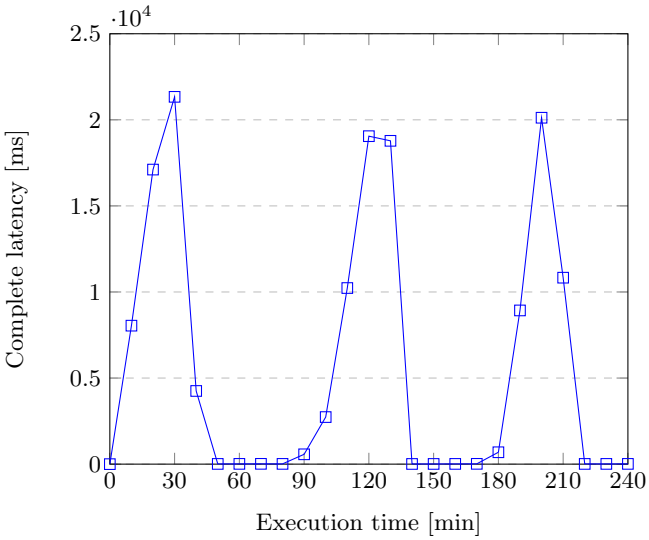

Figure 14: Average complete latency of the tuples emitted by RedisSpout of topology in Fig. 11 over four hours.

7. RELATED WORKS

Many research works in recent years investigated the usage of MDE to support the design and the formal verification of software and embedded systems. [11] presents a systematic literature review on the formal verification of static software models. Most of the works make use of UML models, often enriched with OCL constraints, and only a part of them is fully supported by a tool implementing the model transformations and the verification process. A number of other works have used a model-driven approach for the formal verification of behavioral models (see, e.g., [9, 12]), without addressing the specificities of DIAs. To the best of our knowledge, few works try to address the problem of the verification of DIAs, none of them adopting the MDE approach. They mainly focus on the verification of properties that depend exclusively on the framework by building ad-hoc models; for example, [15] verifies data locality, deadlock freedom and non-termination properties for the Hadoop parallel architecture, while [17] verifies the validity of communication data flows of Hadoop MapReduce. Our work, on the other hand, aims at allowing for the verification of properties that depend on the application design.

8. CONCLUSION

In this paper we presented the model-driven approach to the formal verification of Storm topologies supported by the D-VerT tool. It allows designers to formally check whether, given the features of the components of the topology, it is possible for the queues of some bolts to grow indefinitely, which entails that incoming tuples will not be processed in a timely manner.

Future works will focus on enlarging the set of properties that can be analyzed, on improving the efficiency of the verification technique and on performing an extensive validation on cluster infrastructures.

Acknowledgments

Work supported by Horizon 2020 project no. 644869 (DICE). We are thankful to our colleague Michele Guerriero for his precious advice and expertise in model-driven software engineering.

9. REFERENCES

[1] Apache Storm. http://storm.apache.org/.

[2] DICE Verification Tool (D-VerT). https://github.com/dice-project/DICE-Verification.

[3] Zot. https://github.com/fm-polimi/zot.

[4] S. T. Allen, M. Jankowski, and P. Pathirana. *Storm Applied: Strategies for Real-time Event Processing*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[5] M. Bersani, M. Erascu, F. Marconi, and M. Rossi. DICE verification tool - initial version. Technical report, DICE Consortium, 2016. www.dice-h2020.eu.

[6] M. Bersani, M. Erascu, F. Marconi, and M. Rossi. DICE verification tool - final version. Technical report, DICE Consortium, 2017. www.dice-h2020.eu.

[7] M. M. Bersani, M. Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, 53(2):171–206, 2016.

[8] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. Perez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič. DICE: Quality-driven development of data-intensive cloud applications. In *Proc. of MiSE*, pages 78–83, 2015. www.diceh2020.eu.

[9] Z. Daw and R. Cleaveland. Comparing model checkers for timed UML activity diagrams. *Science of Computer Programming*, 111, Part 2:277–299, 2015.

[10] A. Gómez, M. Guerriero, J. Merseguer, E. di Nitto, and D. A. Tamburri. Design and quality abstractions - initial version. Technical report, DICE Consortium, 2016. www.dice-h2020.eu.

[11] C. A. González and J. Cabot. Formal verification of static software models in MDE: A systematic review. *Information and Software Technology*, 56(8):821 – 838, 2014.

[12] H. Hansen, J. Ketema, B. Luttik, M. Mousavi, and J. van de Pol. Towards model checking executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering*, 6(1):83–90, 2010.

[13] F. M. M. R. M. M. Bersani, M. Erascu. Deliverable 3.6. dice verification tools âĂŞ intermediate version. Technical report, DICE Consortium, 2016. www.dice-h2020.eu.

[14] F. Marconi, M. M. Bersani, M. Erascu, and M. Rossi. Towards the formal verification of data-intensive applications through metric temporal logic. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016*, pages 193–209, 2016.

[15] G. S. Reddy, Y. Feng, Y. Liu, J. S. Dong, S. Jun, and R. Kanagasabai. Towards formal modeling and verification of cloud architectures: A case study on hadoop. In *2013 IEEE Ninth World Congress on Services*, pages 306–311. IEEE, 2013.

[16] C. Stanca. Apache storm topology tuning approach.

[17] W. Su, F. Yang, H. Zhu, and Q. Li. Modeling mapreduce with CSP. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, 2009.