Continuous Architecting of Stream-Based Systems

Marcello Bersani, Francesco Marconi, Damian A. Tamburri *Member, IEEE,*
Andrea Nodari, Pooyan Jamshidi *Member, IEEE*

***Abstract*—Big data architectures have been gaining momentum in recent years. For instance, Twitter uses stream processing frameworks like Storm to analyse billions of tweets per minute and learn the trending topics. However, architectures that process big data involve many different components interconnected via semantically different connectors making it a difficult task for software architects to refactor the initial designs. As an aid to designers and developers, we developed OSTIA (On-the-fly Static Topology Inference Analysis) that allows: (a) visualising big data architectures for the purpose of design-time refactoring while maintaining constraints that would only be evaluated at later stages such as deployment and run-time; (b) detecting the occurrence of common anti-patterns across big data architectures; (c) exploiting software verification techniques on the elicited architectural models. This paper illustrates OSTIA and evaluates its uses and benefits on three industrial-scale case studies.**

***Index Terms*—DevOps; Continuous Architecting; Big Data Streaming Engines; Apache Storm;**

I. Introduction

Big data applications process large amounts of data for the purpose of gaining key business intelligence through complex analytics using machine-learning techniques [19], [31]. These applications are receiving increased attention in the last years given their ability to yield competitive advantage by direct investigation of user needs and trends hidden in the enormous quantities of data produced daily by the average Internet user. According to Gartner [1] business intelligence and analytics applications will remain a top focus for Chief-Information Officers (CIOs) of most Fortune 500 companies until at least 2017-2018. However, the cost of ownership of the systems that process big data analytics are high due to high infrastructure costs, steep learning curves for the different frameworks (such as Apache Storm [20], Apache Spark [2] or Apache Hadoop [3]) involved in designing and developing big data applications and complexities in large-scale architectures and their governance within networked organizations.

In our experience with designing and developing big data architectures, we observed that a key complexity lies in quickly and continuously evaluating the effectiveness of such architectures. Effectiveness, in big data terms, means that the architecture as well as the architecting processes and tools are able to support design, deployment, operation, refactoring and subsequent (re-)deployment of architectures continuously and consistently with runtime restrictions imposed by big data development frameworks. Storm, for example, requires the processing elements to represent a Directed-Acyclic-Graph (DAG). In toy topologies (comprising few components), such constraints can be effectively checked manually, however, when the number of components in such architectures increases to real-life industrial scale architectures, it is enormously difficult to verify even these "simple" structural DAG constraints. We argue that the above notion of architecture and architecting effectiveness can be maintained through continuous architecting of big data applications consistently with a DevOps organisational structure [33], [11]. In the big data domain, continuous architecting means supporting the continuous and incremental improvement of big data architectural designs - e.g., by changing the topological arrangement of architecture elements or any of their properties such as queue lengths - using a constant stream of analyses on running applications as well as platform and infrastructure monitoring. For example, the industrial parter that aided the evaluation of the results in this paper is currently facing the issue of continuously changing and re-arranging their big data stream processing application to several parameters, for example: (a) types of content that need analysis (multimedia images, audios as opposed to news articles and text); (b) types of users that need recommendation (e.g., governments as opposed to single users). Changing and constantly re-arranging an application's architecture requires constant and *continuous architecting* of architecture elements, their interconnection and their visible properties. Also, providing automated support to this continuous architecting exercise, reduces the (re-)design efforts and increases the speed of big data architectures' (re-)deployability by saving the effort of running trial-and-error experiments on expensive infrastructure.

This paper's contribution in support of said continuous architecting is twofold: (a) we elaborate a series of design anti-patterns and algorithmic manipulation techniques that would help designers identify problems in their designs; (b) we outline OSTIA, that stands for: "On-the-fly Static Topology Inference Analysis" - OSTIA allows designers and developers to infer the application architecture through on-the-fly reverse-engineering and architecture recovery [25]. During this inference step, OSTIA analyses the architecture to verify whether it is consistent with development restrictions and/or deployment constraints of the underlying development frameworks (e.g., DAG constraints). To do so, OSTIA hardcodes intimate knowledge on the streaming development framework (Storm, in our case) and its dependence structure in the form of a meta-model [21]. This knowledge is necessary to make sure that elicited topologies are correct, so that models may be used in at least five scenarios: (a) realising an exportable visual representation of the developed topologies; (b) verifying

Pooyan and Andrea are with the Imperial College London, UK. Feel Free to reach Pooyan and Andrea at: [p.jamshidi,a.nodari15]@imperial.co.uk

Damian, Marcello and Francesco are with the Politecnico di Milano, Milan, Italy. Feel Free to reach Damian, Marcello and Francesco at: [marcellomaria.bersani, francesco.marconi, damianandrew.tamburri]@polimi.it

the ordering of the author does not reflect degrees of contribution - every author contributed equally.

structural constraints on topologies that would only become evident during infrastructure setup or runtime operation; (c) verifying the topologies against anti-patterns [32] that may lower performance and limit deployability/executability; (d) manipulate said topologies to elicit non-obvious structural properties such as linearisation or cascading; (e) finally, use topologies for further analysis, e.g., through model verification [12]. In an effort to offer said support in a DevOps fashion, OSTIA was engineered to act as an architecture recovery mechanism that closes the feedback loop between operational data architectures (Ops phase) and their refactoring phase (Dev phase). Currently, OSTIA focuses on Apache Storm, i.e., one of the most famous and established real-time stream processing engines [20], [36]. The core element of Storm, is called *topology*, which represents the architecture of the processing components of the application (from now we use topology and architecture interchangeably).

This paper outlines OSTIA, elaborating major usage scenarios, benefits and limitations. Also, we evaluate OSTIA using case-study research to conclude that OSTIA does in fact provide valuable insights for continuous architecting of streaming-based big data architectures.

The rest of the paper is structured as follows. Section II outlines our research design. Sections III, IV-A and IV-B describe OSTIA, discussing the (anti-)patterns it supports and its usage scenarios. Section V evaluates OSTIA while Section VI discusses results and evaluation outlining OSTIA limitations and threats to validity. Finally, Sections VII and VIII report related work and conclude the paper.

II. Research Design

The work we elaborated in this paper is stemming from the following research question:

"How can we assist the continuous architecting of stream processing systems?"

This research question emerged as part of our work within the DICE EU H2020 project [4] where we evaluated our case-study owners' scenarios and found that their continuous architecting needs were: (a) focusing on the topological abstractions and surrounding architectural specifications; (b) focusing on bridging the gap between insights from Ops to (re-)work at the Dev level; (c) their needs primarily consisted in maintaining framework consistency during architecture reworks. In pursuit of the research question above, the results contained in this paper were initially elaborated within a free-form focus group [27] involving three experienced practitioners and researchers on big data streaming technologies, such as Storm. Following the focus group, through self-ethnography [23] and brainstorming we identified the series of essential consistency checks, algorithmic evaluations as well as anti-patterns that can now be applied through OSTIA while recovering an architectural representation for Storm topologies. We designed OSTIA to support the incremental and iterative refinement of streaming topologies based on the incremental discovery and correction of the above checks and patterns.

Finally, OSTIA's evaluation is threefold. First, we evaluated our solution using an industrial case-study offered by one of the industrial partners in the DICE EU H2020 Project consortium [4]. The partner in question uses open-source social-sensing software to elaborate a subscription-based big-data application that: (a) aggregates news assets from various sources (e.g., Twitter, Facebook, etc.) based on user-desired specifications (e.g., topic, sentiment, etc.); (b) presents and allows the manipulation of data. The application in question is based on the SocialSensor App [5] which features the combined action of three complex streaming topologies based on Apache Storm (see Fig. 1 for a sample realised using a simple UML object diagram). In particular, the topology in Fig. 1 extracts data from sources and manipulates said data to divide and arrange contents based on type (e.g., article vs. media), later updating a series of databases (e.g., Redis) with these elaborations. The models that OSTIA elicited from this application were showcased to our industrial partner in a focus group aimed at establishing the value of insights produced as part of OSTIA-based analyses. Our qualitative assessment was based on questionnaires and open discussion.

Second, to further confirm the validity of OSTIA analyses and support, we applied it on two open-source applications featuring Big-Data analytics and built on top of the Storm streaming technology, namely: (a) the DigitalPebble application, i.e., quoting from the website [6], "A Text Classification API in Java originally developed by DigitalPebble Ltd. The API is independent from the ML implementations used and can be used as a front end to various ML algorithms"; (b) the StormCV application, i.e., quoting from the website [7] "StormCV enables the use of Apache Storm for video processing by adding computer vision (CV) specific operations and data model; the platform enables the development of distributed video processing pipelines which can be deployed on Storm clusters".

Third, finally, we applied well-established verification approaches to integrate the value and benefits behind using OSTIA. We engineered OSTIA to support exporting of elicited topologies for their further analysis using the Zot [22] LTL model-checker using an approach outlined in our previous work [12], [13].

III. Research Solution

This section outlines OSTIA starting form a brief recap of the technology it is currently designed to support, i.e., the Apache Storm framework. Further on, the section introduces how OSTIA was designed to support continuous architecting of streaming topologies focusing on Storm. Finally, the section outlines the meta-model for Storm that captures all restrictions and rules (e.g., for configuration, topology, dependence, messaging, etc.) in the framework. OSTIA uses this meta-model as a reference every time the application is run to recover and analyse operational topologies.

A. OSTIA design

The overall architecture of OSTIA is depicted in Figure 2. The logical architectural information of the topology is retrieved by OSTIA via static analysis of the source code.

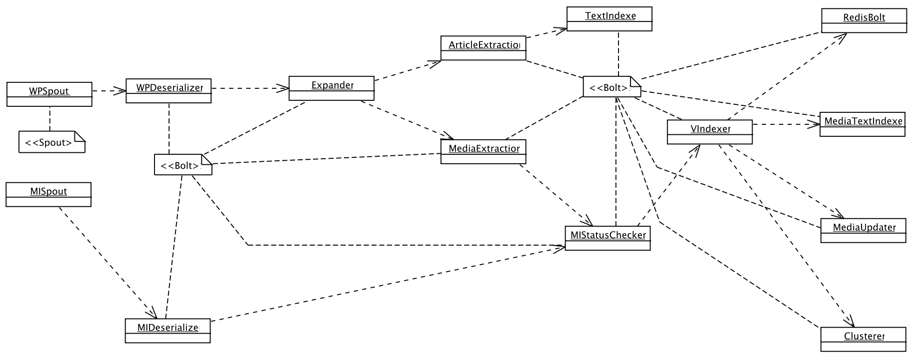


Fig. 1. A sample Storm topology (readable from left to right) using an UML object diagram in the SocialSensor App, notes identify types for nodes (i.e., Bolts or Spouts).

OSTIA generates a simple intermediate format to be used by other algorithmic processes.

OSTIA is architected in a way that algorithmic analysis, such as anti-pattern analyses, can be easily added. These analyses use the information resides in the intermediate format and provide added value analyses for continuous architecting of storm topologies. Since the information in the intermediate format only rely on the logical code analysis, the algorithmic analyses require some information regarding the running topology, such as end to end latency and throughput.

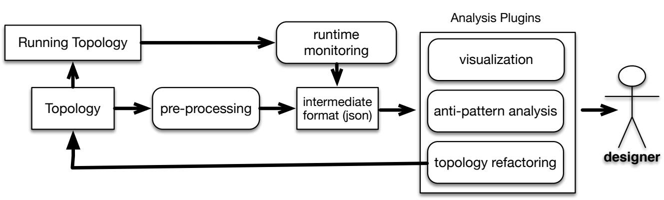


Fig. 2. OSTIA extensible architecture.

Such information will be continuously added to the intermediate repository via runtime monitoring of the topology on real deployment cluster. These provide appropriate and rich information for refactoring the initial architecture and enabling performance driven DevOps [14]. Finally, OSTIA allows users to export the topology in different formats (e.g. JSON) to analyse and continuously improve the topology with other tools, e.g., by means of formal verification.

B. Storm Architecture

Storm is a technology developed at Twitter [36] in order to face the problem of processing of streaming of data. It is defined as a distributed processing framework which is able to analyse streams of data. The core element in the system is called *topology*. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts. The former type includes nodes that process the data entering the topology, for instance querying APIs or retrieve information from a message broker, such as Apache Kafka. The latter executes operations on data, such as filtering or serialising.

C. Storm Framework Meta-Model

OSTIA was designed to retrieve and analyse Storm topologies on-the-fly, allowing their refactoring in a way which is consistent with framework restrictions, rules and regulations part of the Storm framework. To do so, OSTIA uses a meta-model for the Storm framework which acts as an operational image of all said restrictions and rules that OSTIA needs to maintain. Essentially OSTIA uses the meta-model as such an operational image for Storm, for two purposes: (a) checking that Storm restrictions (e.g., Spouts initiate the topology) and constraints (e.g., grouping policies) are valid on models recovered by OSTIA; (b) keep checking said restrictions and constraints during continuous architecting. The meta-model in question is depicted in Fig. 3. The figure shows an overview of the meta-model for Storm[1] where, for example, the grouping restrictions that Storm envisions are captured in an enumeration of constraints (see the <<Grouping>> element or the <<ReplicationFactor>> concrete parameter). Key elements of the meta-model are the following:

- <<TopologyConfiguration>> contains the parameters necessary for the Storm framework to be configured and to run on the selected infrastructure. OSTIA checks that these parameters are present or that defaults are correctly inplace;
- <<Topology>> specifies the topological construct being elicited for the analysed Storm application, as composed of the <<Bolt>> and the <<Spout>> meta-elements;
- <<Grouping>> contains restrictions on the possible groupings of the <<Bolt>> and the <<Spout>> meta-elements within the elicited topology. OSTIA checks these restrictions upon recovery and exporting of topologies;

D. Storm: A Formal Interpretation

Model-checking can serve as a means to enact continuous architecting of Storm topologies. Topologies can undergo formal verification, for example, to assess temporal properties on their execution. This section elaborates on the role of

[1]The details of this meta-model and the restrictions captured therein is beyond the scope of this paper. More details are available here: http://dice-h2020.eu/deliverables/D2.1.

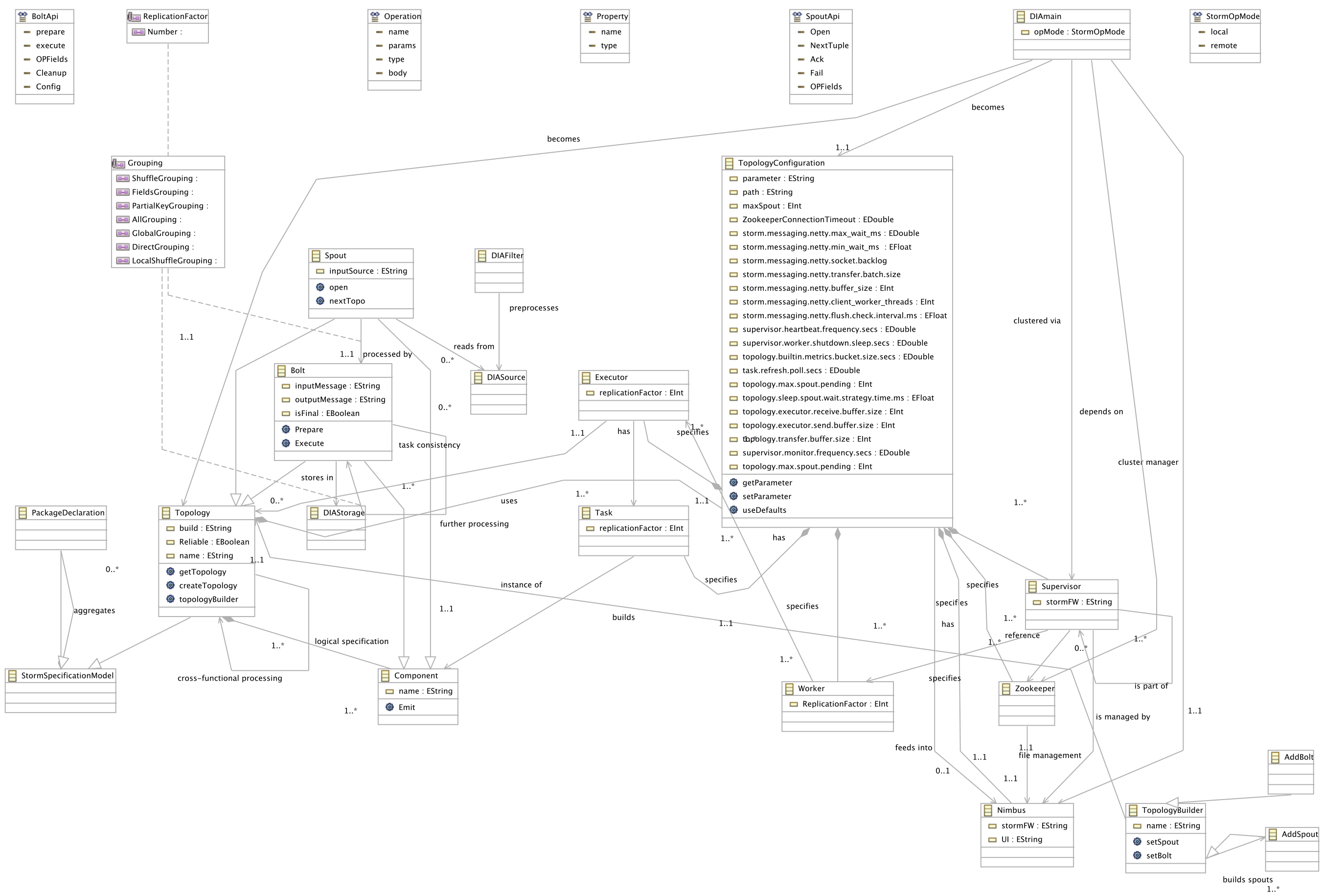


Fig. 3. The Storm Meta-Model.

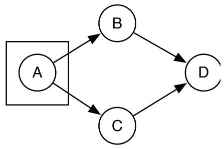
formal verification in OSTIA and describes the necessary background, modelling assumptions and model definition behind Storm topology verification. In particular, we provide a non-deterministic model representing Storm topologies' behavior in terms of the delay connected to bolts' processing, spout input profile and node failures. Spout input profile is measured with rates of incoming tuples into the topology. Verification in OSTIA is intended to discover possible design errors at design time which are caused by (i) under/over estimation of timing requirements of computational nodes or (ii) possible runtime node failures. Therefore, in this context, we are interested in verifying properties like, for instance, the existence of an execution of the topology which guarantees queue-length boundedness even if failures occur with a certain delay. Defining the formal model, requires the comprehension of the behaviors of both spouts and bolts which, after choosing the level of abstraction of the model, allows us to abstract those behaviors accordingly, to formalize them as finite state machines. The purpose of this activity is defining the operations performed by nodes and their allowed orderings in a real implementation. We then extend the model considering the message buffers (or queues) and the quantity of tuples that are exchanged through the topology. In addition, we introduce more specific temporal constraints to limit the time spent by the system in each state (or processing phase) and to elaborate the concept of *rate*, intended as "number of times an event is occurring every time unit". The formal modeling (see Section IV-D) is based on real-time temporal logic, i.e., the topology behavior is defined through a temporal logic formula written in Constraint LTL over clocks (CLTLoc) [13].

IV. OSTIA-Based Continuous Architecting

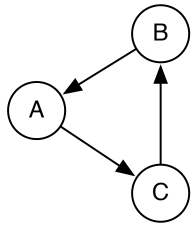
This section elaborates on the ways in which OSTIA supports continuous architecting. First, we elaborate on the anti-patterns supported in OSTIA. Second, we elaborate on the algorithmic manipulation that OSTIA could apply to topologies to provide alternative visualisation. Third, we discuss how OSTIA suggests an alternative architecture to improve the system performance. Fourth, finally, we elaborate on how OSTIA can drive continuous improvement assisted by formal verification. All figures in these sections use a simple graph-like notation where nodes may be any topological element (e.g., Spouts or Bolts in Apache Storm terms) while edges are directed data-flow connections.

A. Topology Design Anti-Patterns Within OSTIA

This section elaborates on the anti-patterns we elicited through self-ethnography. These anti-patterns are elaborated further within OSTIA to allow for their detection during



(a) Multi-anchoring.



(b) Cycle-in.

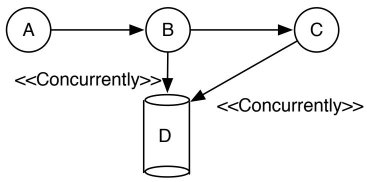


Fig. 4. Concurrency management in case of Persistent Data circumstances.

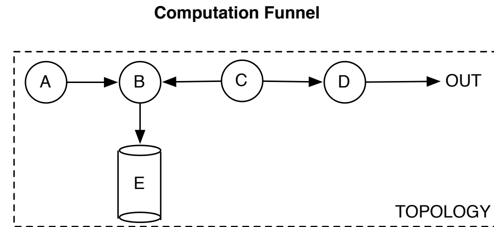


Fig. 5. computation funnel.

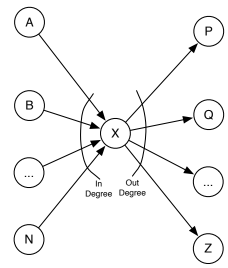


Fig. 6. Fan-in fan-out in Stream topologies.

streaming topology inference analysis. Every pattern is elaborated using a simple graph-like notation where *spouts* are nodes that have outgoing edges only whereas *bolts* are nodes that can have either incoming or outgoing edges respectively.

1) Multi-Anchoring: The Multi-Anchoring pattern is shown in Fig. IV-A1. In order to guarantee fault-tolerant stream processing, tuples processed by bolts need to be anchored with the unique id of the bolt and be passed to multiple acknowledgers (or "ackers" in short) in the topology. In this way, ackers can keep track of tuples in the topology. Multiple ackers can indeed cause much overhead and influence the operational performance of the entire topology.

2) Cycle-in Topology: The Cycle-in pattern is shown in Fig. IV-A1. Technically, it is possible to have cycle in Storm topologies. An infinite cycle of processing would create an infinite tuple tree and make it impossible for Storm to ever acknowledge spout emitted tuples. Therefore, cycles should be avoided or resulting tuple trees should be investigated additionally to make sure they terminate at some point and under a specified series of conditions. The anti-pattern itself may lead to infrastructure overloading and increased costs.

3) Persistent Data: The persistent data pattern is shown in Fig. 4. This pattern covers the circumstance wherefore if two processing elements need to update a same entity in a storage, there should be a consistency mechanism in place. OSTIA offers limited support to this feature, which we plan to look into more carefully for future work. More details on this support are discussed in the approach limitations section.

4) Computation Funnel: The computational funnel is shown in Fig. 5. A computational funnel emerges when there is not a path from data source (spout) to the bolts that sends out the tuples off the topology to another topology through a messaging framework or through a storage. This circumstance should be dealt with since it may compromise the availability of results under the desired performance restrictions.

B. Algorithmic Analysis on Stream Topologies

This section elaborates on the algorithmic analysis supported by OSTIA using the common graph-like notation introduced previously. OSTIA currently supports two topology content analysis (see Sec. IV-B1 and IV-B2) as well as two topology layout analyses (see Sec. IV-B3 and IV-B4). Only a part of these analyses is currently implemented in OSTIA. We discuss approach limitations further in Sect. VI.

1) Fan-in/Fan-out: The Fan-in/Fan-out algorithmic manipulation is outlined in Fig. 6. For each element of the topology, fan-in is the number of incoming streams. Conversely, fan-out is the number outgoing streams. In the case of bolts, both in and out streams are internal to the topology. For Spouts, incoming streams are the data sources of the topology (e.g., message brokers, APIs, etc).

This algorithmic manipulation allows to visualise instances where fan-in and fan-out numbers are differing.

2) Topology cascading: The Cascading algorithmic manipulation is outlined in Fig. 7. By topology cascading, we mean connecting two different Storm topologies via a messaging framework (e.g., Apache Kafka [8]). Although cascading may simplify the development of topologies by encouraging architecture elements' reuse especially for complex but procedural topologies, this circumstance may well raise the complexity of continuous architecting and may require separation of concerns [26]. For example, Fig. 7 outlines an instance in which two topologies are concatenated together by a message broker. In this instance, formal verification may be applied on the left-hand side topology, which is more business-critical, while the right-hand side of the entire topology is improved by on-the-fly OSTIA-based analysis. Even though OSTIA support for this feature is still limited, we report it nonetheless since OSTIA was engineered to address multiple topologies at once.

This algorithmic manipulation allows to combine multiple

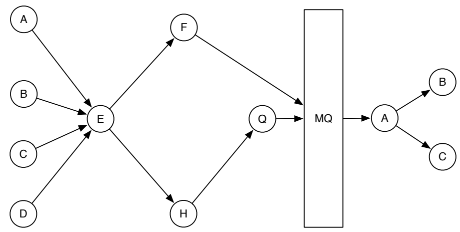


Fig. 7. cascading.

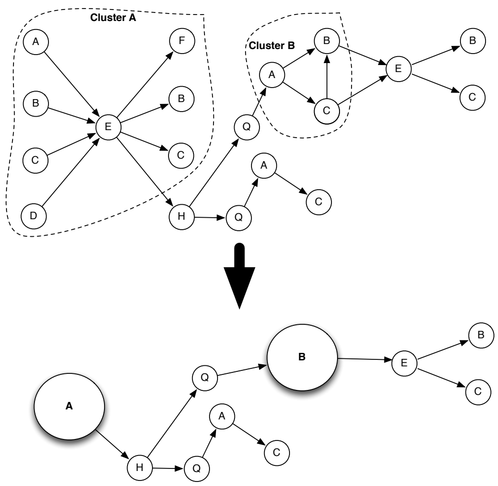


Fig. 8. clustering.

cascading topologies.

3) Topology clustering: Topology clustering is outlined in Fig. 8. Topology clustering implies identifying coupled processing elements (i.e., bolts and spouts) and cluster them together (e.g., by means of graph-based analysis) in a way that elements in a cluster have high cohesion and loose-coupling with elements in other clusters. Simple clustering or Social-Network Analysis mechanisms can be used to infer clusters. These clusters may require additional attention since they could turn out to become bottlenecks. Reasoning more deeply on clusters and their resolution may lead to establishing the Storm scheduling policy best-fitting with the application.

4) Linearising a topology: Topology linearisation is outlined in Fig. 9. Sorting the processing elements in a topology in a way that topology looks more linear, visually. This step ensures that visual investigation and evaluation of the structural complexity of the topology is possible by direct observation. It is sometimes essential to provide such a visualisation to evaluate how to refactor the topology as needed.

C. Suggestions for Performance Improvements

Big data architectures typically need parameters tuning to achieve best performance. For instance, in Storm developers have to specify the parallelism level for each node, which is the number of processes instantiated for a particular bolt or spout. OSTIA provides suggestions on how to change the parallelism level of the nodes, using simple and fast heuristics together with static analysis.

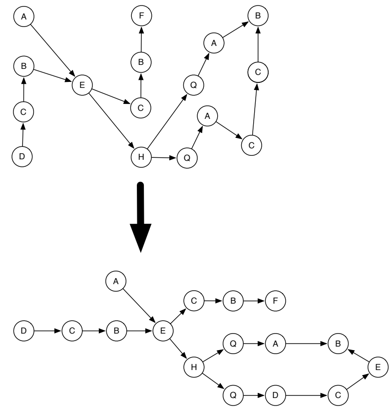


Fig. 9. linearising.

After architects designed a Storm application, a scheduler instantiate the topology on a cluster of machines. The default scheduler utilises a round-robin strategy to fairly load the machines with the bolts and spouts. This a crucial assumption for the heuristic used in OSTIA to perform well. There has been some proposal to change the default scheduler logic **[Andrea I added this sentence but please add some citations here]** in order to boost the performance of the topologies, however, many Storm users typically prefer the default scheduler while having the opportunity to tune the parameter automatically behind the scenes.

The heuristic works as follow. A Storm architect runs OSTIA specifying the number of machines used in the deployment and the number of instances of spouts and bolts that can be spawned in each machine. OSTIA statically analyses the topology and extract the parallelism level for each component of the topology. At this point, we sum of all instances need to be allocated and the slots available on the machines ($machines * components\_for\_each\_machine$).

OSTIA decides whether a possible improvements is possible (i.e. *slots available* $>$ *instances to be allocate*), and suggests changes to the parallelism level to the nodes in order to improve the performance. The simplest case occur when the unallocated slots are enough to remove a machine from the cluster, thus saving costs.

Alternatively, OSTIA identify a subset of nodes, called critical nodes, which are important from a performance perspective. The critical nodes of a topology are defined as the nodes with the highest *weighted fan-in*. The *weighted fan-in* of a node N is defined by Equation 1.

$$\text{weighted fan-in(N)} = \frac{\sum_{X \to N \in Edges} parallelism(X)}{parallelism(N)} \quad (1)$$

The critical nodes could be easily susceptible to overloading as their parallelism level do not compensate the parallelism level of its *in-nodes*. Increasing the parallelism level gives the nodes more resources to cope with high loads.

For instance, let us take Figure 11 as an example. There are 22 components that needs to be allocated. Suppose that our cluster is composed by 4 machines and each machine fits 10 instances of components. OSTIA in this case would suggest to simply remove one machine. Let us suppose that we have 3 machines with 10 tasks each. At this point we have 30 slots available and 22 components, therefore we have 8 slots available that can be used to increase the performance. In order to decide the components to improves we identify the ones with maximum *weighted fan-in*. In the example nodes *mediaExtractor* and *articleExtractor* with *weighted fan-in* of 8. Finally, since we have 8 free slots to share between two nodes we increase the parallelism level of the two critical nodes of $8/2 = 4$, setting it from 1 to 5.

[I really liked the section but I think if we can find a way to link this up with the verification somehow and evaluate this would be perfect, can Francesco do this?]

D. OSTIA-Based Formal Verification

This section describes the formal modelling and verification employed in OSTIA. Our assumption for continuous architecting, is that architects eliciting and studying their topologies by means of OSTIA may want to continuously and incrementally improve it based on results from solid verification approaches. The approach we outline, relies on *satisfiability checking* [28], an alternative approach to model-checking where, instead of an operational model (like automata or transition systems), the system (i.e., a topology in this context) is specified by a formula defining its executions over time and properties are verified by proving that the system logically entails them.

CLTLoc is a real-time temporal logic and, in particular, a semantic restriction of Constraint LTL (CLTL) [18] allowing atomic formulae over $(\mathbb{R}, \{<, =\})$ where the arithmetical variables behave like clocks of Timed Automata (TA) [30]. A clock x measures the time elapsed since the last time when $x = 0$ held, i.e., since the last "reset" of x. Clocks are interpreted over Reals and their value can be tested with respect to a positive integer 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:

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

where $x \in X$, $y, z \in Y$, $c \in \mathbb{N}$ and $\sim \in \{<, =\}$, $\bullet$, $\circ$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since". A *model* is a pair (π, σ), where σ is a mapping associating every variable x and position in $\mathbb{N}$ with value $\sigma(i, x)$ and π is a mapping associating each position in $\mathbb{N}$ with subset of AP. 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 $\sim$ than/to c and formula $\mathrm{X}y \sim z \pm c$ states that the next value of variable y is $\sim$ to/than $z + c$.

The standard technique to prove the satisfiability of CLTL and CLTLoc formulae is based on of Büchi automata [18], [13] but, for practical implementation, Bounded Satisfiability Checking (BSC) [28] avoids the onerous construction of automata by means of a reduction to a decidable Satisfiability Modulo Theory (SMT) problem [13]. The outcome of a BSC problem is either an infinite ultimately periodic model or unsat.

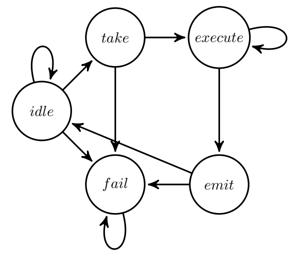


Fig. 10. Finite state automaton describing bolt states.

CLTLoc allows the specification of non-deterministic models using temporal constraints wherein clock variables range over a dense domain and whose value is not abstracted. Clock variables represent, in the logical language and with the same precision, physical (dense) clocks implemented in real architectures. Clocks are associated with specific events to measure time elapsing over the executions. As they are reset when the associated event occurs, in any moment, the clock value represents the time elapsed since the previous reset and corresponds to the elapsed time since the last occurrence of the event associated to it. We use such constraints to define, for instance, the time delay required to process tuples or between two node failures.

Building on top of the above framework, a formal interpretation of the Storm (meta-)model requires the following abstractions and assumptions:

- some deployment details, e.g., the number of worker nodes and features of the underlying cluster, are abstracted;
- each bolt/spout has a single output stream;
- there is a single queuing layer: every bolt has a unique incoming queue and no sending queue, while the worker queues are not represented;
- every operation is performed within minimum and maximum thresholds of time;
- the content of the messages is not relevant: all the tuples have the same fixed size and we represent only quantity of tuples moving through the system;

A Storm Topology is a directed graph $\mathbf{G} = \{\mathbf{N}, Sub\}$ where the set of nodes $\mathbf{N} = \mathbf{S} \bigcup \mathbf{B}$ includes in the sets of spouts ($\mathbf{S}$) and bolts ($\mathbf{B}$) and $Sub \subset \mathbf{N} \times \mathbf{N}$ defines how the nodes are connected each other via the subscription relation. Pair $(i, j) \in Sub$ indicates that "bolt i subscribes to the streams emitted by the spout/bolt j". Spouts cannot subscribe to other nodes in the topology. Each bolt has a receive queue where the incoming tuples are collected before being read and processed. The queues have infinite size and the level of occupation of each j^{th} queue is described by the variable q_j. Spouts have no queues, and each spout can either *emit* tuples into the topology or stay *idle*. Each bolt can be in *idle* state, in *failure* state or in *processing* state. While in the processing state, the bolt first reads tuples from its receive queue (*take* action), then it performs its transformation (*execute* action) and finally it *emits* the output tuples in its output streams.

We provide, as an example, one of the formulae defining the processing state. Formula 2 can be read as *"for all bolts: if a bolt j is processing tuples, then it has been processing tuples since it took those tuples from the queue, (or since the origin of the events), and it will keep processing those tuples until it will either emit them or fail. Moreover, the bolt is not in a failure state"*.

$$\bigwedge_{i \in \mathbf{B}} \left(\begin{array}{l} \mathtt{process}_i \Rightarrow \\ \mathtt{process}_i \, \mathbf{S} \, (\mathtt{take}_i \vee (\mathtt{orig} \wedge \mathtt{process}_i)) \wedge \\ \mathtt{process}_i \, \mathbf{U} \, (\mathtt{emit}_i \vee \mathtt{fail}_i) \wedge \neg \mathtt{fail}_i \end{array} \right) \tag{2}$$

The number of tuples emitted by a bolt depends on the number of incoming tuples. The ratio $\frac{\#output\_tuples}{\#input\_tuples}$ expresses the "kind of function" performed by the bolt and is given as configuration parameter. All the emitted tuples are then added to the receive queues of the bolts subscribing to the emitting nodes. In the same way, whenever a bolt reads tuples from the queue, the number of elements in queue decreases. To this end, formula 3, imposes that *"if a bolt takes elements from its queue, the number of queued elements in the next time instant will be equal to the current number of elements plus the quantity of tuples being added (emitted) from other connectd nodes minus the quantity of tuples being read"*.

$$\bigwedge_{j \in B} (\mathtt{take}_j \Rightarrow (\mathrm{X} q_j = q_j + r_{\mathtt{add}_j} - r_{\mathtt{take}_j})) \tag{3}$$

These functional constraints are fixed for all the nodes and they are not configurable. The structure of the topology, the parallelism level of each node, the bolt function and the non-functional requirements, as, for example, the time needed for a bolt in order to process a tuple, the minimum and maximum time between failures and the spout emitting rate are configurable parameters of the model. Currently, the verification tool accepts a JSON file containing all the configuration parameters. OSTIA supports such format and is able to extract from static code analysis a partial set of features, and an almost complete set of parameters after monitoring a short run of the system. The user can complete the JSON file by adding some verification-specific settings.

V. Evaluation

We evaluated OSTIA through qualitative evaluation and case-study research featuring an open-/closed-source industrial case study (see Section V-A) and two open-source case studies (see Section V-B) on which we also applied complex formal verification (see Section IV-D).

A. Industrial Case-Study

OSTIA was evaluated using several topologies part of the SocialSensor App. Our industrial partner is having performance and availability outages connected to currently unknown circumstances. Therefore, the objective of our evaluation for OSTIA was twofold: (a) allow our industrial partner to enact continuous architecting of their application with the goal of discovering any patterns or hotspots that may be requiring further architectural reasoning; (b) understand whether OSTIA provided valuable feedback to endure the continuous architecting exercise.

OSTIA standard output[2] for the smallest of the three SocialSensor topologies, namely the "focused-crawler" topology, is outlined in Fig. 11.

OSTIA has been proved particularly helpful in visualising the complex topology together with the parallelism level of each components. Combining this information with runtime data, such as latency, our industrial partner observed that the "expander" bolt needed additional architectural reasoning. Also, the partner welcomed the idea of using OSTIA as a mechanism to enact continuous architecting of the topology in question as part of the needed architectural reasoning.

Besides this pattern-based evaluation and assessment, OSTIA algorithmic analyses assisted our client in understanding that the topological structure of the SocialSensor app would be better fit for batch processing rather than streaming, since the partner observed autonomously that too many database-output spouts and bolts were used in their versions of the SocialSensor topologies. In so doing, the partner is now using OSTIA to drive the refactoring exercise towards a Hadoop Map Reduce [3] framework for batch processing.

B. Evaluation on Open-Source Software

To confirm the usefulness and capacity of OSTIA to enact a continuous architecting cycle, we applied it in understanding (first) and attempting improvements of two open-source applications, namely, the previously introduced DigitalPebble [9] and StormCV [7] applications. Figures 13 and 12 outline standard OSTIA output for the two applications. Note that we did not have any prior knowledge concerning the two applications in question and we merely run OSTIA on the applications' codebase dump in our own experimental machine. OSTIA output takes mere seconds for small to medium-sized topologies (e.g., around 25 nodes).

The OSTIA output aided as follows: (a) the output summarised in Fig. 13 allowed us to immediately grasp the functional behavior of the DigitalPebble and StormCV topologies allowing us to interpret correctly their operations before reading long documentation or inspecting the code; (b) OSTIA aided us in visually interpreting the complexity of the applications at hand; (c) OSTIA allowed us to spot several anti-patterns in the DigitalPebble Storm application around the "sitemap" and "parse" bolts, namely, a multiple cascading instance of the multi-anchoring pattern and a persistent-data pattern. Finally, OSTIA aided in the identification of the computational funnel anti-pattern around the "status" bolt closing the DigitalPebble topology. With this evaluation at hand, developers in the respective communities of DigitalPebble and StormCV could refactor their topologies, e.g., aided by OSTIA-based formal verification that proves the negative effects of said anti-patterns.

C. Continuous Architecting by Means of Formal Verification: An Industrial Case-Study

In this section we outline the results from OSTIA-based formal verification applied on (one of) the topologies used

[2]Output of OSTIA analyses is not evidenced for the sake of space.

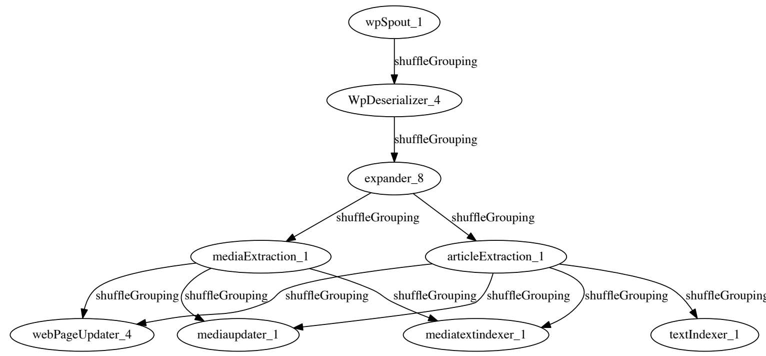


Fig. 11. SocialSensor App, OSTIA sample output.

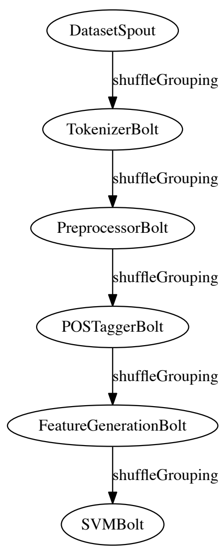


Fig. 12. StormCV topology.

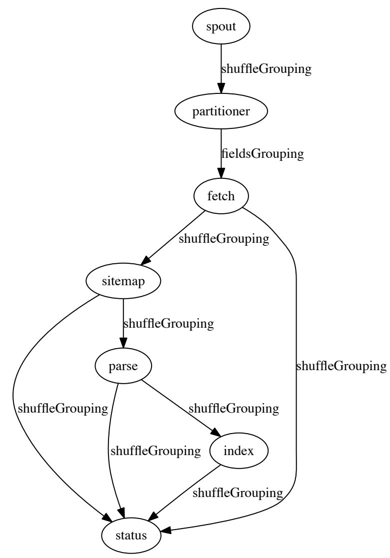


Fig. 13. DigitalPebble topology.

by our industrial partner in practice. Results provide valuable insights for (re-)architecting and improving these topologies in a continuous manner.

The formal analysis of the “focused-crawler” topology confirmed the critical role of the “expander” bolt, previously noticed with the aim of OSTIA visual output. It emerged from the output traces that there exists an execution of the system, even without failures, where the queue occupation level is unbounded. Figure 14 shows how the tool constructed a periodic model in which a suffix (highlighted in red) of a finite sequence of events is repeated infinitely many times after a prefix (in white). After ensuring that the trace is not a spurious model, we concluded that the expander queue, having an increasing trend in the suffix, is unbounded.

VI. Discussion

This section discusses some findings and the limitations of OSTIA.

A. Findings and Continuous Architecting Insights

OSTIA represents one humble, but significant step at supporting practically the necessities behind developing and maintaining high-quality big-data application architectures. In designing and developing OSTIA we encountered a number of insights that may aid continuous architecting.

First, we found (and observed in industrial practice) that it is often more useful to develop a quick-and-dirty but “runnable” architecture topology than improving the topology at design time for a tentatively perfect execution. This is mostly the case with big-data applications that are developed stemming from previously existing topologies or applications.

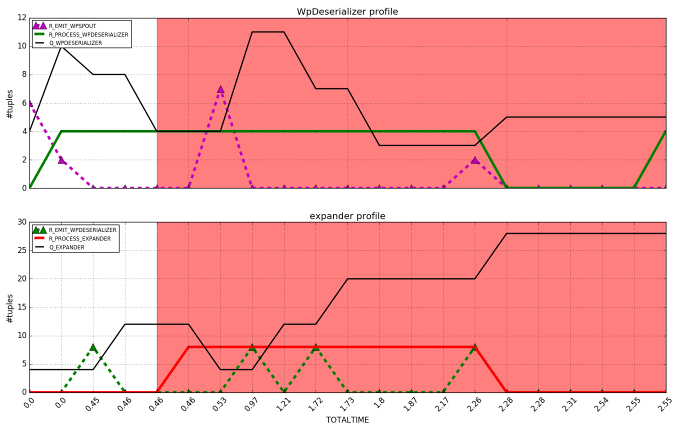


Fig. 14. OSTIA-based formal verification output trace showing the evolution of the two bolts over time. Queue trends are displayed as solid black line. Green and red solid line show the processing activity of the bolts, while the dashed lines illustrate the incoming tuples from the subscribed nodes (`emit` events).

OSTIA hardcodes this way of thinking by supporting reverse-engineering and recovery of deployed topologies for their incremental improvement. Although we did not carry out extensive qualitative or quantitative evaluation of OSTIA in this regard, we are planning additional industrial experiments for future work with the goal of increasing OSTIA usability and practical quality.

Second, big-data applications design is an extremely young and emerging field for which not many software design patterns have been discovered yet. The (anti-)patterns and approaches currently hardcoded into OSTIA are inherited from related fields, e.g., pattern- and cluster-based graph analysis. Nevertheless, OSTIA may also be used to investigate the existence of recurrent and effective design solutions (i.e., design patterns) for the benefit of big-data application design. We are improving OSTIA in this regard by experimenting on two fronts: (a) re-design and extend the facilities with which OSTIA supports anti-pattern detection; (b) run OSTIA on multiple big-data applications stemming from multiple technologies beyond Storm (e.g., Spark, Hadoop Map Reduce, etc.) with the purpose of finding recurrent patterns. A similar approach may feature OSTIA as part of architecture trade-off analysis campaigns [17].

Third, a step which is currently undersupported during big-data applications design is devising an efficient algorithmic breakdown of a workflow into an efficient topology. Conversely, OSTIA does support the linearisation and combination of multiple topologies, e.g., into a cascade. Cascading and similar super-structures may be an interesting investigation venue since they may reveal more efficient styles for big-data architectures beyond styles such as Lambda Architecture [29]. OSTIA may aid in this investigation by allowing the interactive and incremental improvement of multiple (combinations of) topologies together.

B. Approach Limitations and Threats to Validity

Although OSTIA shows promise both conceptually and as a practical tool, it shows several limitations.

First of all, OSTIA only supports streaming topologies enacted using Storm. Multiple other big-data frameworks such as Apache Spark, Samza exist to support both streaming and batch processing.

Second, OSTIA only allows to recover and evaluate previously-existing topologies, its usage is limited to design improvement and refactoring phases rather than design. Although this limitation may inhibit practitioners from using our technology, the (anti-)patterns and algorithmic approaches elaborated in this paper help designers and implementors to develop the reasonably good-quality and "quick" topologies upon which to use OSTIA for continuous improvement.

Third, OSTIA does offer essential insights to aid deployment as well (e.g., separating or *clustering* complex portions of a topology so that they may run on dedicated infrastructure), however, our tool was not meant to be used as a system to aid planning and infrastructure design. Rather, as specified previously in the introduction, OSTIA was meant to evaluate and increase the quality of topologies *before* they enter into operation since the continuous improvement cycles connected to operating the topology and learning from said operation are often costly and still greatly inefficient.

Fourth, although we were able to discover a number of recurrent anti-patterns to be applied during OSTIA analysis, we were not able to implement all of them in practice and in a manner which allows to spot both the anti-pattern and any problems connected with it. For example, detecting the "Cycle-in topology" is already possible, however, OSTIA would not allow designers to understand the consequence of the anti-pattern, i.e., where in the infrastructure do the cycles cause troubles. Also, there are several features that are currently under implementation but not released within the OSTIA codebase, for example, the "Persistent Data" and the "Topology Cascading" features.

In the future we plan to tackle the above limitations furthering our understanding of streaming design as well as the support OSTIA offers to designers during continuous architecting.

VII. Related Work

The work behind OSTIA stems from the EU H2020 Project called DICE [4] where we are investigating the use of model-driven facilities to support the design and quality enhancement of big data applications. Much similarly to the DICE effort, the IBM Stream Processing Language (SPL) initiative [24] provides an implementation language specific to programming streams management (e.g., Storm jobs) and related reactive systems. In addition, there are several work close to OSTIA in terms of their foundations and type of support.

First, from a quantitative perspective, much literature discusses quality analyses of Storm topologies, e.g., from a performance [37] or reliability point of view [34]. Existing work use complex math-based approaches to evaluating a number of big data architectures, their structure and general configuration. However, these approaches do not suggest any architecture refactorings. With OSTIA, we automatically elicits a Storm topology, analyses the topologies against a number of consistency constraints that make the topology consistent with the framework. To the best of our knowledge, no such tool exists to date.

Second, from a modelling perspective, approaches such as StormGen [16] offer means to develop Storm topologies in a model-driven fashion using a combination of generative techniques based on XText and heavyweight (meta-)modelling, based on EMF, the standard Eclipse Modelling Framework Format. Although the first of its kind, StormGen merely allows the specification of a Storm topology, without applying any consistency checks or without offering the possibility to *recover* said topology once it has been developed. By means of OSTIA, designers can work refining their Storm topologies, e.g., as a consequence of verification or failed checks through OSTIA. Tools such as StormGen can be used to assist preliminary development of quick-and-dirty topologies.

Third, from a verification perspective, to the best of our knowledge, this represents the first attempt to build a formal model representing Storm topologies, and the first try in making a configurable model aiming at running verification tasks of non-functional properties for big data applications. While some works concentrate on exploiting big data technologies to speedup verification tasks [15], others focus on the formalization of the specific framework, but remain application-independent, and their goal is rather to verify properties of the framework, such as reliability and load balancing [35], or the validity of the messaging flow in MapReduce [38].

VIII. Conclusion

We set out to assist the continuous architecting of big data streaming designs by OSTIA, a toolkit to assist designers and developers to facilitate static analysis of the architecture and provide automated constraint verification in order to identify design anti-patterns and provide structural refactorings. OSTIA helps designers and developers by recovering and analysing the architectural topology on-the-fly, assisting them in: (a) reasoning on the topological structure and how to refine it; (b) export the topological structure consistently with restrictions of their reference development framework so that further analysis (e.g., formal verification) may ensue. In addition, while performing on-the-fly architecture recovery, the analyses that OSTIA is able to apply focus on checking for the compliance to essential consistency rules specific to targeted big data frameworks. Finally, OSTIA allows to check whether the recovered topologies contain occurrences of key anti-patterns. By running a case-study with a partner organization, we observed that OSTIA assists designers and developers in establishing and continuously improving the quality of topologies behind their big data applications. We confirmed this result running OSTIA on several open-source applications featuring streaming technologies. We released OSTIA as an open-source software [10].

In the future we plan to further elaborate the anti-patterns, exploiting graphs analysis techniques inherited from social-networks analysis. Also, we plan to expand OSTIA to support further technologies beyond the most common application framework for streaming, i.e., Storm. Finally, we plan to further evaluate OSTIA using empirical evaluation.

Acknowledgment: The Authors' work is partially supported by the European Commission grant no. 644869 (EU H2020), DICE. Also, Damian's work is partially supported by the European Commission grant no. 610531 (FP7 ICT Call 10), SeaClouds.

Appendix: Please follow the link to navigate to the appendices: http://tinyurl.com/zco4sdz.

References

[1] [Online]. Available: http://www.gartner.com/newsroom/id/2637615
[2] [Online]. Available: http://spark.apache.org/
[3] [Online]. Available: https://hadoop.apache.org/
[4] [Online]. Available: http://www.dice-h2020.eu/
[5] [Online]. Available: https://github.com/socialsensor
[6] [Online]. Available: https://github.com/DigitalPebble/storm-crawler
[7] [Online]. Available: https://github.com/sensorstorm/StormCV
[8] [Online]. Available: http://kafka.apache.org/
[9] [Online]. Available: https://github.com/DigitalPebble
[10] [Online]. Available: https://github.com/maelstromdat/OSTIA
[11] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective.*, ser. SEI series in software engineering. Addison-Wesley, 2015.
[12] M. M. Bersani, S. Distefano, L. Ferrucci, and M. Mazzara, "A timed semantics of workflows." in *ICSOFT (Selected Papers)*, ser. Communications in Computer and Information Science, vol. 555. Springer, 2014, pp. 365–383.
[13] M. M. Bersani, M. Rossi, and P. San Pietro, "A tool for deciding the satisfiability of continuous-time metric temporal logic," *Acta Informatica*, pp. 1–36, 2015.
[14] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski *et al.*, "Performance-oriented devops: A research agenda," *arXiv preprint arXiv:1508.04752*, 2015.
[15] M. Camilli, "Formal verification problems in a big data world: Towards a mighty synergy," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 638–641.
[16] K. Chandrasekaran, S. Santurkar, and A. Arora, "Stormgen - a domain specific language to create ad-hoc storm topologies." in *FedCSIS*, 2014, pp. 1621–1628.
[17] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.
[18] S. Demri and D. D'Souza, "An automata-theoretic approach to constraint LTL," *Information and Computation*, vol. 205, no. 3, pp. 380–415, 2007.
[19] C. K. Emani, N. Cullot, and C. Nicolle, "Understandable big data: A survey." *Computer Science Review*, vol. 17, pp. 70–81, 2015.
[20] R. Evans, "Apache storm, a hands on tutorial." in *IC2E*. IEEE, 2015, p. 2.
[21] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002.
[22] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, "Modeling time in computing: A taxonomy and a comparative survey," *ACM Comput. Surv.*, vol. 42, no. 2, pp. 6:1–6:59, Mar. 2010.
[23] M. Hammersley and P. Atkinson, *Ethnography*. London: Routledge, 2003.
[24] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. P. Mendell, H. Nasgaard, S. Schneider, R. Soul, and K.-L. Wu, "Ibm streams processing language: Analyzing big data in motion." *IBM Journal of Research and Development*, vol. 57, no. 3/4, p. 7, 2013.
[25] I. Ivkovic and M. W. Godfrey, "Enhancing domain-specific software architecture recovery." in *IWPC*. IEEE Computer Society, 2003, pp. 266–273.
[26] T. Mens and M. Wermelinger, "Issue overview: Separation of concerns in software evolution," vol. 14, no. 5, pp. 311–315, 2002.
[27] D. L. Morgan, *Focus groups as qualitative research*. Sage Publications, Thousand Oaks, 1997.
[28] M. Pradella, A. Morzenti, and P. S. Pietro, "Bounded satisfiability checking of metric temporal logic specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 20:1–20:54, Jul. 2013.
[29] M. Quartulli, J. Lozano, and I. G. Olaizola, "Beyond the lambda architecture: Effective scheduling for large scale eo information mining and interactive thematic mapping." in *IGARSS*, 2015, pp. 1492–1495.
[30] Rajeev Alur and David L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, p. 183?235, 1994.

[31] B. Ratner, *Statistical and Machine-Learning Data Mining: Techniques for Better Predictive Modeling and Analysis of Big Data*. CRC PressINC, 2012.

[32] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture. Vol. 2, Patterns for concurrent and networked objects*. John Wiley, Chichester, England, 2000.

[33] D. A. Tamburri, P. Lago, and H. van Vliet, "Organizational social structures for software engineering." *ACM Comput. Surv.*, vol. 46, no. 1, p. 3, 2013.

[34] Y. Tamura and S. Yamada, "Reliability analysis based on a jump diffusion model with two wiener processes for cloud computing with big data." *Entropy*, vol. 17, no. 7, pp. 4533–4546, 2015.

[35] M. M. Tommaso Di Noia and E. D. Sciascio, "A computational model for mapreduce job flow," pp. 335–340, 2014.

[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[37] D. Wang and J. Liu, "Optimizing big data processing performance in the public cloud: opportunities and approaches." *IEEE Network*, vol. 29, no. 5, pp. 31–35, 2015.

[38] F. Yang, W. Su, H. Zhu, and Q. Li, "Formalizing mapreduce with csp," in *Proceedings of ECBS*. IEEE Computer Society, 2010, pp. 358–367.

PLACE PHOTO HERE

Marcello Maria Bersani The italian formalist crazy for logic.

PLACE PHOTO HERE

Damian A. Tamburri The canadian-italian crazy motherfucker.

PLACE PHOTO HERE

Francesco Marconi Marcello's wingman.

PLACE
PHOTO
HERE

Pooyan Jamshidi the crazy iranian who is a sword with Storm and loves cats.

PLACE
PHOTO
HERE

Andrea Nodari the italian englishman who was crazy enough to implement OSTIA.