Supporting Continuous Architecting of Data-Intensive Applications

Marcello M. Bersani, Francesco Marconi, Damian A. Tamburri

Politecnico di Milano, Milan, Italy

[*marcellomaria.bersani, francesco.marconi, damianandrew.tamburri*]*@polimi.it*

Andrea Nodari, Pooyan Jamshidi

[*p.jamshidi,a.nodari15*]*@imperial.co.uk*

[a]*Imperial College London, UK*

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.

Keywords: DevOps, Continuous Architecting, Big Data

1. Introduction

Big data or *data-intensive* applications (DIAs) process large amounts of data for the purpose of gaining key business intelligence through complex analytics using machine-learning techniques [1, 2]. 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 [3] 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 infrastructure costs, steep learning curves for the different frameworks (such as Apache Storm [4], Apache Spark [5] or Apache Hadoop [6]) typically involved in design and development of big data applications.

A key complexity of the above design and development activity lies in quickly and continuously refining the configuration parameters of of the middleware and service platforms on top which the DIA is running - we define this process as *continuous architecting* [7]. The process in question, namely, continuous architecting, is especially complex as the middleware in DIAs increases, since the many big data middleware involved have each over 100+ parameters (e.g., latency or beaconing times, caching policies, queue retention and more) - *fine-tuning these "knobs" on so many concurrent technologies requires an automated tool to speed up this heavily manual, trial-and-error continuous fine-tuning process*.

The entry-point for such fine-tuning is the DIA's graph of operations, which we call *topology* - a topology[1] is a directed graph which represents the cascade of operations to be applied on data in a batch (i.e., slicing the data and analysing one partition at the time with the same operations) or stream (i.e., continuous data analysis) processing fashion. The topology can either be known or directly extracted from DIA code.

This paper offers two major contributions in support of DIAs continuous architecting: (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 recover the application architecture topology applying on-the-fly reverse-engineering and automated architecture recovery [8]; (c) an OSTIA extension for formal verification of temporal properties evaluated over the reverse-engineered topology.

First, during its reverse-engineering 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., constraints). To do so, OSTIA hardcodes intimate knowledge on key big data processing frameworks (Apache Storm and Apache Hadoop2, in our case) and their dependency structure in the form of a meta-model

[1] the term is slightly overridden from the classical notion of topologies existing in DIAs, e.g., Apache Storm topologies.

[9]. This knowledge is necessary to infer from data-intensive source-code topologies are correct and correctly configured.

Second, OSTIA-elicited models may be used in at least five scenarios: (a) realising an exportable visual representation of the developed topologies; (b) verifying, the structural constraints on topologies that would only become evident during infrastructure setup or runtime operation; (c) verifying the topologies against anti-patterns [10] 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 in a completely transparent fashion and thanks to formal verification techniques [11].

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). As previously stated, currently, OSTIA focuses on Apache Hadoop and Apache Storm, i.e., the most famous and established real-time batch and stream processing engines [4, 12], respectively.

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. Although a previous version of this paper was published in the proceedings of WICSA 2015 [7], we introduce the following novel contributions:

- we extended OSTIA to address Apache Hadoop Architectures and re-executed the evaluation in line with this addition;
- we extended OSTIA with a formal verification feature for elicited topologies, using a formal model built via formal CLTLoc - this feature operates LTL Constraint verification over-clocks and is completely transparent to OSTIA users, checking autonomously for safety of OSTIA-elicited topologies;
- we extended OSTIA with 3 heuristics that expert big data practitioners in the WICSA 2015 working group panel suggested as valuable additions;

The rest of the paper is structured as follows. The next section elaborates further on the emerging notion of continuous architecting for DIAs. Section 3 outlines our research design and context of study. Section 4 outlines OSTIA. Section 6 evaluates OSTIA while Section 7 discusses results and evaluation outlining OSTIA limitations and threats to validity. Finally, Sections 8 and 9 report related work and conclude the paper.

2. Terminological Reflections on Continuous Architecting

From a software architecture perspective, topologies are not properly *software architectures*. A software architecture is a set of interconnected and inter-communicating components across a series of connectors and other architecture elements [13]. Rather, DIA topologies are based on operations to be applied on the raw data. For the sake of designing our tool, however, we inherited from software architecture research and software architecture recovery tools and practices with which we enable DIA topology reverse-engineering.

On one hand, DIA topologies bear the same weight on overall DIA performance and application properties that regular software architectures reflect for regular applications.

On the other hand, multiple middleware with different topologies may be merged within the same DIA. In the scope of OSTIA, the tool makes not difference between topology and architecture, since the tool recuperates all distinct topology graphs it finds in the namespace provided as input.

We this premise, *we advocate that an analogy exists between the classical notion of software architecture and our notion of DIA topology - for this reason, from now on, we use the terms topology and architecture interchangeably.*

To evaluate DIA effectiveness in big data terms we mean that the architecture as well as the architecting processes and tools are able to support design, deployment, operation, refactoring, and subsequent (re-)deployment of DIAs continuously and consistently with runtime restrictions imposed by big data development frameworks. Storm, for example, is an Apache big data processing middleware which 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.

In the big data domain, such **continuous** architecting means supporting the continuous and incremental improvement of big data topologies - e.g., by: (1) changing the topological arrangement of architecture elements or any of their properties (e.g., queues and their lengths); (2) using on-the-fly analyses on running applications; (3) exploiting platform and infrastructure monitoring data. For example, the industrial partner that aided the evaluation of the results in this paper is currently facing the issue of continuously changing and re-arranging their stream processing application. Particularly, they require to change in response to: (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 DIAs, their interconnection and their visible properties.

With OSTIA, we aimed at providing automated support to this continuous architecting exercise. OSTIA reduces the (re-)design efforts in that exercises and increases the speed of big data architectures' (re-)deployability by saving the effort of running trial-and-error experiments on expensive infrastructure.

3. Research Design and Context of Study

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 [14] 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 architectural consistency during refactoring. In pursuit of the research question above, the results contained in this paper were initially elaborated within 3 structured focus-groups [15] involving 5 experienced practitioners and researchers on big data technologies, such as Storm. The practitioners were simply required to elaborate on the most frequent structural issues they encountered on their DIA design experiments.

We designed OSTIA to support the incremental and iterative refinement of streaming topologies based on the incremental discovery and correction of the anti-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 [14]. 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 [16] 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 divides and arranges 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, namely: (a) the DigitalPebble application, "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" [17]; (b) the StormCV application, "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" [18].

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 further analyses using the Zot [19] LTL model-checker using an approach outlined in our previous work [11, 20].

4. Research Solution

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

4.1. OSTIA Tool Architecture

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. 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. 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.

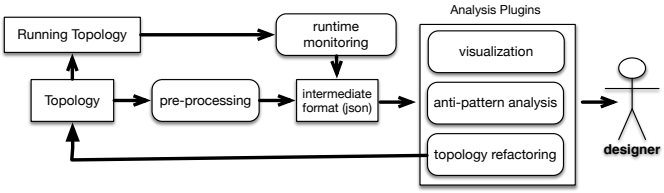


Figure 2: OSTIA extensible architecture.

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 [21]. 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.

4.2. A Concrete Example: The Storm Architecture

Storm is a technology developed at Twitter [12] 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. 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

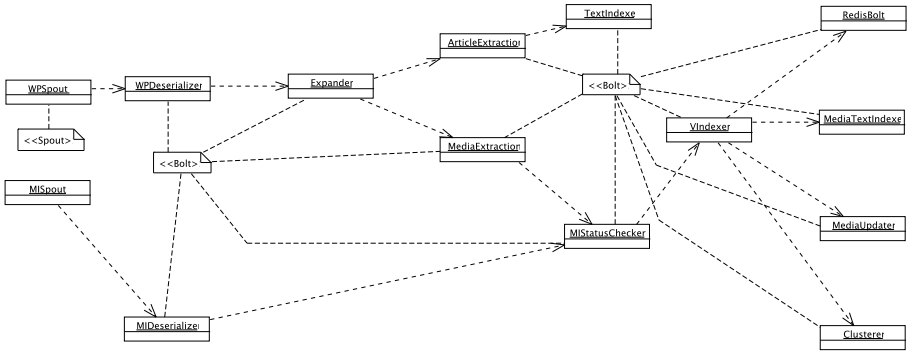


Figure 1: A sample Storm topology (readable from left to right) using an UML object diagram in the SocialSensor App, notes identify types for nodes (i.e., Bolts or Spouts).

from a message broker, such as Apache Kafka[2]. The latter executes operations on data, such as filtering or serialising.

4.2.1. 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. To give a hint of the complexity of the technology, we outline the meta-model in Fig. 3. 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;

A complete overview of the details of this meta-model and the restrictions captured therein is beyond the scope of this paper - rather, the entire purpose of OSTIA is to hide their complexity: for example, notice the *TopologyConfiguration* meta-class, where we deliberately selected 22 (about 10% of the entire set) of parameters possibly configurable for running Storm topologies. Further detail on the Storm meta-model may be found on the full technical report describing our technology[3].

4.2.2. 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 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 5.4) 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) [20].

[2] `http://kafka.apache.org/`

[3] `http://dice-h2020.eu/deliverables/D2.1`

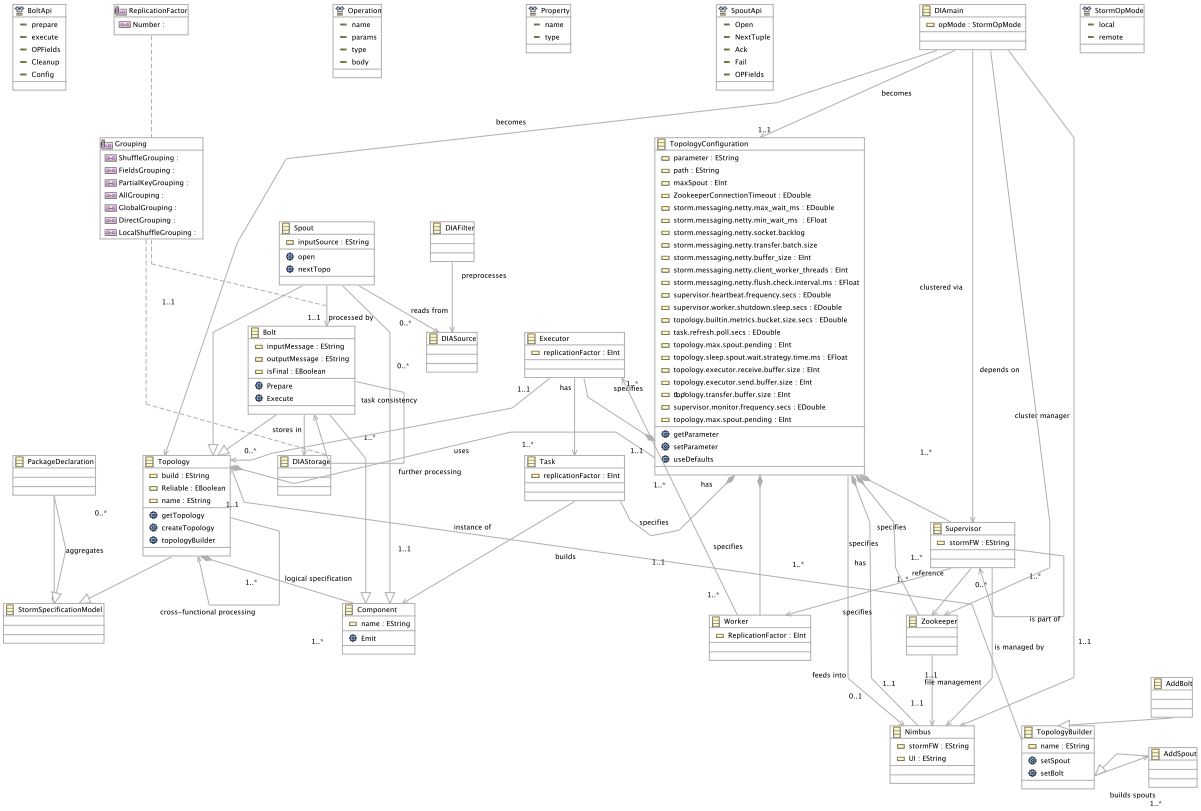


Figure 3: The Storm Meta-Model, an overview.

5. OSTIA-Based Continuous Architecting

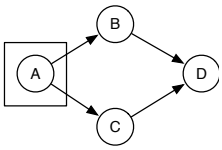
This section elaborates on the ways in which OSTIA supports continuous architecting. First, we elaborate on the anti-patterns supported in OSTIA. Second, we elaborate on the algorithmic analyse-and-refactor actions that OSTIA can apply to topologies to provide alternative visualisation and improved topology structure. Third, we discuss how OSTIA suggests an alternative architecture to improve the system performance. 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.

5.1. Topology Design Anti-Patterns Within OSTIA

This section elaborates on the anti-patterns we elicited (See Section 3). These anti-patterns are elaborated further within OSTIA to allow for their detection during 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.

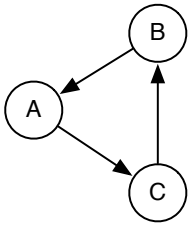
5.1.1. Multi-Anchoring

The Multi-Anchoring pattern is shown in Fig. 4. 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.



(a) Multi-anchoring.

Figure 4: The multi-anchoring anti-pattern.



(a) Cycle-in.

Figure 5: The cycle-in anti-pattern.

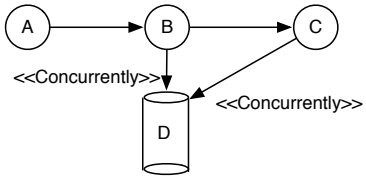


Figure 6: Concurrency management in case of Persistent Data circumstances.

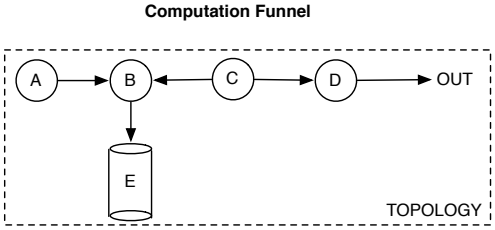


Figure 7: computation funnel.

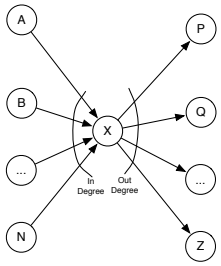


Figure 8: Fan-in fan-out in Stream topologies.

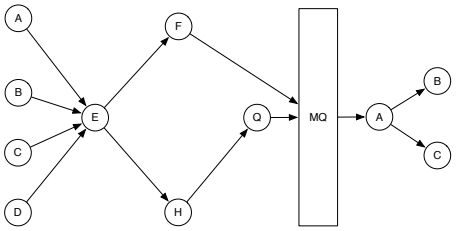


Figure 9: cascading.

5.1.2. Cycle-in Topology

The Cycle-in pattern is shown in Fig. 5. 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 (these conditions can be hardcoded in Bolt logic). The anti-pattern itself may lead to infrastructure overloading and increased costs.

5.1.3. Persistent Data

The persistent data pattern is shown in Fig. 6. 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.

5.1.4. Computation Funnel

The computational funnel is shown in Fig. 7. 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.

5.2. algorithmic analyse-and-refactor actions on Stream Topologies

This section elaborates on the algorithmic analyse-and-refactor actions supported by OSTIA using the common graph-like notation introduced previously. OSTIA currently supports two topology content analysis (see Sec. 5.2.1 and 5.2.2) as well as two topology layout analyses (see Sec. 5.2.3 and 5.2.4). Only a part of these analyses is currently implemented in OSTIA. We discuss approach limitations further in Sect. 7.

5.2.1. Fan-in/Fan-out

The Fan-in/Fan-out algorithmic analyse-and-refactor action is outlined in Fig. 8. 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) which live outside of the topology.

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

5.2.2. Topology cascading

The Cascading algorithmic analyse-and-refactor action is outlined in Fig. 9. By topology cascading, we mean connecting two different Storm topologies via a messaging framework (e.g., Apache Kafka [22]). 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 [23]. For example, Fig. 9 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.

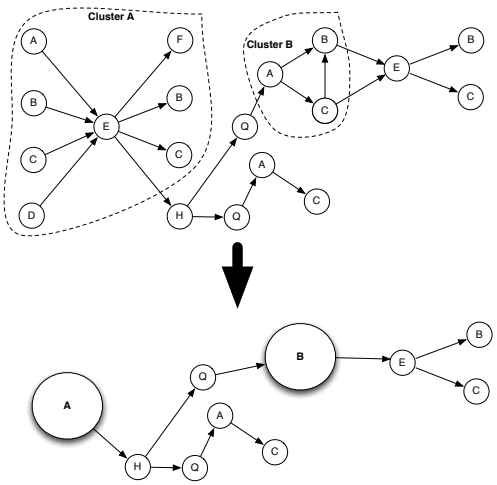


Figure 10: clustering.

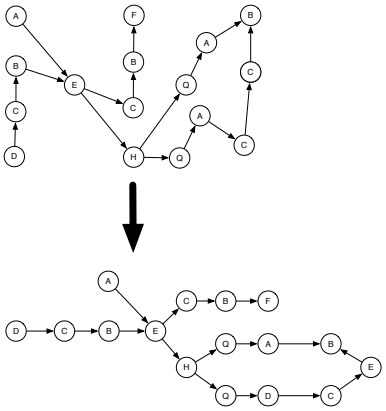


Figure 11: linearising.

This algorithmic action allows to combine multiple cascading topologies.

5.2.3. Topology clustering

Topology clustering is outlined in Fig. 10. 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. We will elaborate on this in Section 5.3.

5.2.4. Linearising a topology

Topology linearisation is outlined in Fig. 11. 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.

5.3. Performance Improvement Heuristics

In this section, we elaborate on a specific case where algorithmic analyse-and-refactor actions improve the performance of the data-intensive application. More in particular, 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.

After architects designed a Storm application, a scheduler instantiates 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 are several proposals in the state of the art to change the default scheduler logic in order to boost the performance of the topologies [24, 25]. However, many DIA users typically prefer the default scheduler, while having the opportunity to tune the parameters automatically behind the scenes.

The OSTIA heuristic works as follows: A DIA 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 extracts 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 occurs when the unallocated slots are enough to remove a machine from the cluster, thus saving costs.

Alternatively, OSTIA identifies 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 13 as an example. There are 22 components that need 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.

The above heuristic approach concludes the support that OSTIA offers to quickly and continuously improving streaming topologies by means of algorithmic evaluations and analysis. Conversely, as previously stated we learned from industrial practice that the need might rise for more formal guarantees, e.g., concerning some key parts of a streaming topology with respect to key big-data properties such as queue-boundedness - i.e., a guarantee that the queue for a certain bolt will always be bounded - or queue-population - i.e., a guarantee that the queue never contains privacy sensitive objects. In similar scenarios, OSTIA offers the seamless capability of running complex and exhaustive formal verification over analysed topologies. The next section elaborates further on this key support offered by OSTIA.

5.4. 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, which was first proposed in [26], relies on *satisfiability checking* [27], 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) [28] allowing atomic formulae over $(\mathbb{R}, \{<, =\})$ where the arithmetical variables behave like clocks of Timed Automata (TA) [29]. As for TA, clocks measures time delays between events: 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 or reset to 0. To analyse anomalous executions of Storm topologies which do not preserve the queue-length boundedness property for the nodes of the application, we consider CLTLoc with counters. Counters are discrete non-negative variables that are used in our model to represent the length of bolt queues over the time throughout the streaming processing realized by the application. 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 $\sim \in \{<, =\}$, $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{U}$ 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 [28, 20] but, for practical implementation, Bounded Satisfiability Checking (BSC) [27] avoids the onerous construction of automata by means of a reduction to a decidable Satisfiability Modulo Theory (SMT) problem [20]. The outcome of a BSC problem is either an infinite ultimately periodic model or unsat.

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, in [26] we provide a formal interpretation of the Storm (meta-)model which requires several abstractions and assumptions.

- key deployment details, e.g., the number of worker nodes and features of the underlying cluster, are abstracted away;
- 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 ($\mathbf{S}$)

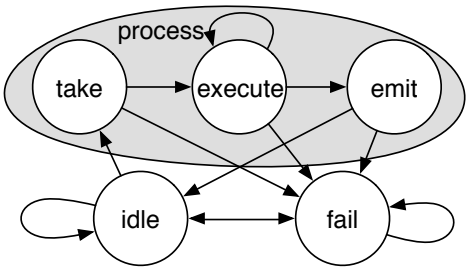


Figure 12: Finite state automaton describing bolt states.

and bolts (**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.

An excerpt of the full model designed in [26] is shown in Fig. 12. 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"*.

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

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"*.

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

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.

6. Evaluation

We evaluated OSTIA through qualitative evaluation and case-study research featuring an open-/closed-source industrial case study (see Section 6.1) and two open-source case studies (see Section 6.2) on which we also applied complex formal verification (see Section 5.4).

6.1. 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[4] for the smallest of the three SocialSensor topologies, namely the "focused-crawler" topology, is outlined in Fig. 13.

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 (i.e., latency times) our industrial partner observed that the "expander" bolt needed additional architectural reasoning. More in particular, the bolt in question concentrates a lot of the topology's progress on its queue, greatly hampering the topology's scalability. In our partner's scenario, the limited scalability was blocking the expansion of the topology in question with more data sources and sinks. In addition, 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

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

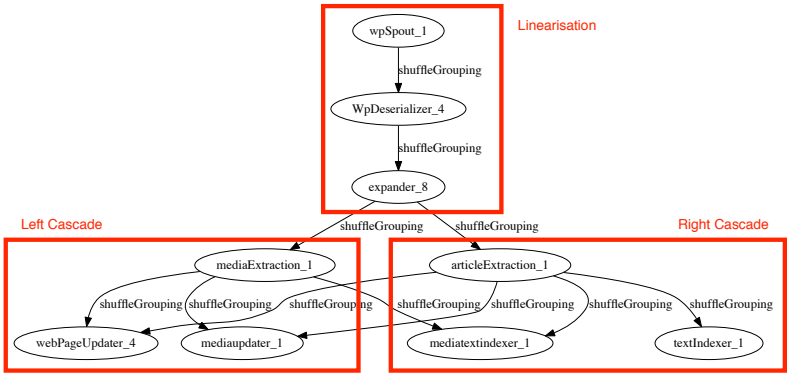


Figure 13: SocialSensor App, OSTIA sample output partially linearised (top) and cascaded (bottom left and right).

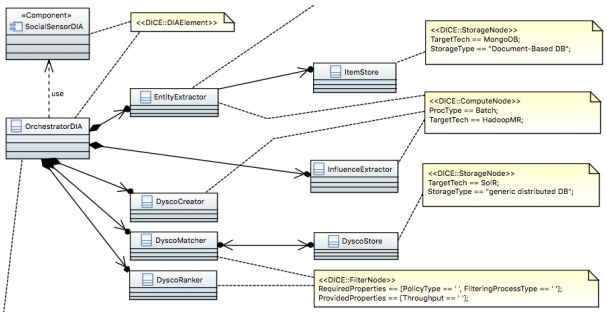


Figure 14: Industrial case-study, a refactored architecture.

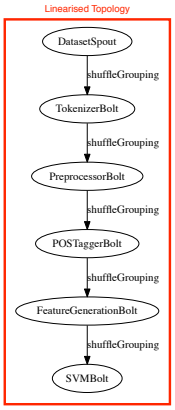


Figure 15: StormCV topology (linearised).

to drive the refactoring exercise towards a Hadoop Map Reduce [6] framework for batch processing.

As a followup of our analysis, our partner is refactoring his own high-level software architecture adopting a lambda-like software architecture style [30] (see Fig. 14) which includes the Social-Sensor App (Top of Fig. 14) as well as several additional computation components. In summary, the refactoring resulting from OSTIA-based analysis equated to deferring part of the computations originally intended in the expander bolt part of the Storm topology within the Social Sensor app to additional ad-hoc Hadoop Map Reduce jobs with similar purpose (e.g., the EntityExtractor compute node in Fig. 14) and intents but batched out of the topological processing in Storm (see Fig. 14)[5].

Our qualitative evaluation of the refactored architecture by means of several interviews and workshops revealed very encouraging results. However, we are yet to quantitatively evaluate wether the new software architecture actually reflects a tangible boost in terms of performance and scalability.

6.2. *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 [31] and StormCV [18] applications. Figures 16 and 15 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).

[5] several other overburdened topological elements were refactored but were omitted here due to industrial secrecy

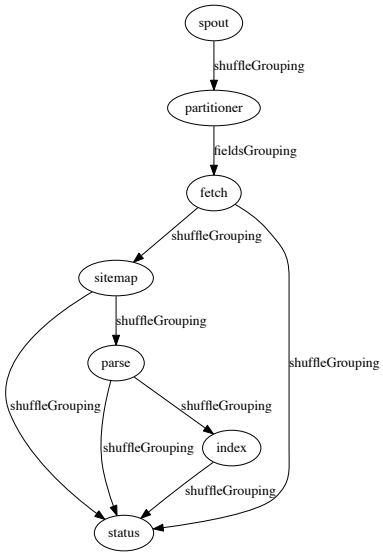


Figure 16: DigitalPebble topology.

The OSTIA output aided as follows: (a) the output summarised in Fig. 16 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.

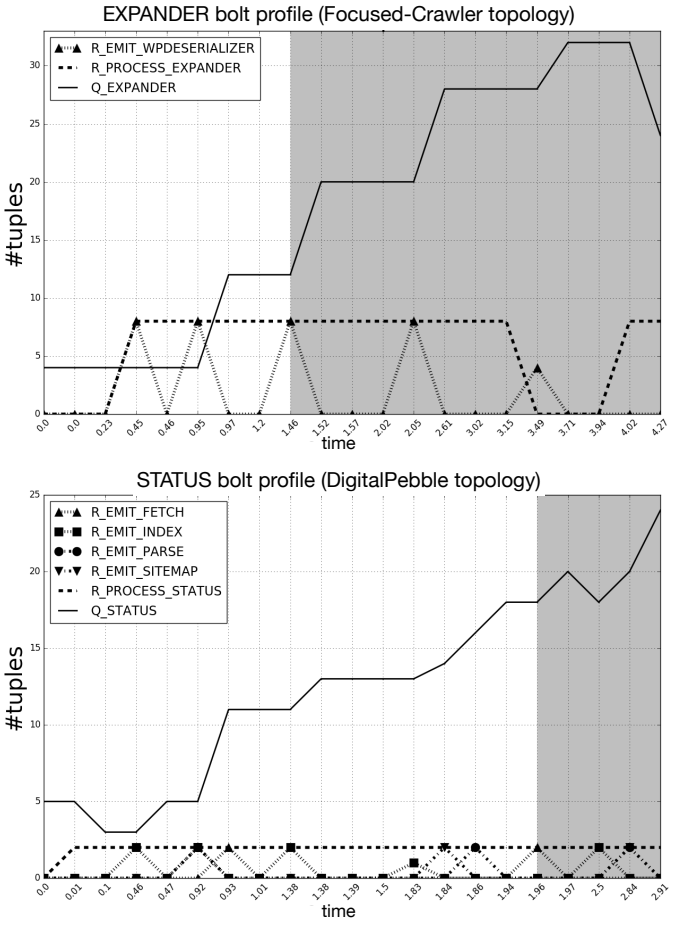


Figure 17: OSTIA-based formal verification output traces showing the evolution of the two bolts over time. Queue trends are displayed as solid black line. Dashed lines show the processing activity of the bolts, while the other lines illustrate the incoming tuples from the subscribed nodes (`emit` events).

6.3. 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 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 of the bolt is unbounded. Figure 17 shows how the tool constructed a periodic model in which a suffix (highlighted by the gray background) of a finite sequence of events is repeated infinitely many times after a prefix (on white background). 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. As shown in the the output trace at the bottom of Fig. 17, further analyses on the DigitalPebble use case revealed that the same problem affects the "status" bolt of the DigitalPebble topology. This finding from the formal verification tool reinforced the outcome of the anti-pattern module of OSTIA, showing how the presence of the computational funnel anti-pattern could lead to an unbounded growth in the queue of the "status" bolt. These types of heavyweight and powerful analyses are made easier by OSTIA in that our tool provides a ready-made analyzable counterpart of the elicited topologies making almost invisible the formal verification layer (other than manually setting and tuning operational parameters for verification) on top of which OSTIA support is harnessed.

7. Discussion

This section discusses some findings and the limitations of OSTIA.

7.1. 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. OSTIA hardcodes this way of thinking by supporting reverse-engineering and recovery of deployed topologies for their incremental improvement. Such improvement is helpful because these topologies running continuously on rented clusters and the refactoring can help in boosting the performance and therefore requiring less resources and less cost for the rented clusters. 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., Apache 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 [32].

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 [30] and Microservices [33]. OSTIA may aid in this investigation by allowing the interactive and incremental improvement of multiple (combinations of) topologies together.

7.2. 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 only a limited set of DIA middleware technologies. 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) and therefore the tool may serve for the additional purpose of aiding deployment design. However, our tool was not designed to be used as a system that aids deployment planning and infrastructure design. Further research should be invested into combining on-the-fly technology such as OSTIA with more powerful solvers that determine infrastructure configuration details and similar technological tuning, e.g., the works by Peng et Al. [34] and similar.

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.

8. Related Work

The work behind OSTIA stems from the EU H2020 Project called DICE [14] 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 [35] 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 [36] or reliability point of view [37]. 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 [38] 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 [39], 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 [40], or the validity of the messaging flow in MapReduce [41].

9. 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 [42].

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.

[1] C. K. Emani, N. Cullot, C. Nicolle, Understandable big data: A survey., Computer Science Review 17 (2015) 70–81.

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

[3] [link].
URL http://www.gartner.com/newsroom/id/2637615

[4] R. Evans, Apache storm, a hands on tutorial., in: IC2E, IEEE, 2015, p. 2.

[5] [link].
URL http://spark.apache.org/

[6] [link].
URL https://hadoop.apache.org/

[7] M. M. Bersani, F. Marconi, D. A. Tamburri, P. Jamshidi, A. Nodari, Continuous Architecting of Stream-Based Systems, in: H. Muccini, E. K. Harper (Eds.), Proceedings of the 25th IFIP / IEEE Working Conference on Software Architectures, IEEE Computer Society, 2016, pp. 131–142.

[8] I. Ivkovic, M. W. Godfrey, Enhancing domain-specific software architecture recovery., in: IWPC, IEEE Computer Society, 2003, pp. 266–273.

[9] D. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, John Wiley & Sons, Inc., New York, NY, USA, 2002.

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

[11] M. M. Bersani, S. Distefano, L. Ferrucci, M. Mazzara, A timed semantics of workflows., in: ICSOFT (Selected Papers), Vol. 555 of Communications in Computer and Information Science, Springer, 2014, pp. 365–383.

[12] 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.

[13] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, SEI series in software engineering, Addison-Wesley, 2012.
URL https://books.google.com/books?id=-II73rBDXCYC

[14] [link].
URL http://www.dice-h2020.eu/

[15] D. L. Morgan, Focus groups as qualitative research, Sage Publications, Thousand Oaks, 1997.

[16] [link].
URL https://github.com/socialsensor

[17] [link].
URL https://github.com/DigitalPebble/storm-crawler

[18] [link].
URL https://github.com/sensorstorm/StormCV

[19] C. A. Furia, D. Mandrioli, A. Morzenti, M. Rossi, Modeling time in computing: A taxonomy and a comparative survey, ACM Comput. Surv. 42 (2) (2010) 6:1–6:59.

[20] M. M. Bersani, M. Rossi, P. San Pietro, A tool for deciding the satisfiability of continuous-time metric temporal logic, Acta Informatica (2015) 1–36doi:10.1007/s00236-015-0229-y.

[21] 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.

[22] [link].
URL http://kafka.apache.org/

[23] T. Mens, M. Wermelinger, Issue overview: Separation of concerns in software evolution 14 (5) (2002) 311–315.

[24] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in storm, in: Proceedings of the 7th ACM International Conference on Distributed

Event-based Systems, DEBS '13, ACM, New York, NY, USA, 2013, pp. 207–218. doi:10.1145/2488222.2488267.
URL http://doi.acm.org/10.1145/2488222.2488267

[25] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R. Campbell, R-storm: Resource-aware scheduling in storm, in: Proceedings of the 16th Annual Middleware Conference, Middleware '15, ACM, New York, NY, USA, 2015, pp. 149–161. doi:10.1145/2814576.2814808.
URL http://doi.acm.org/10.1145/2814576.2814808

[26] F. Marconi, M. M. Bersani, M. Erascu, M. Rossi, Towards the formal verification of dia through mtl models, in: booktitle, Lecture Notes in Computer Science.

[27] M. Pradella, A. Morzenti, P. S. Pietro, Bounded satisfiability checking of metric temporal logic specifications, ACM Trans. Softw. Eng. Methodol. 22 (3) (2013) 20:1–20:54. doi:10.1145/2491509.2491514.

[28] S. Demri, D. D'Souza, An automata-theoretic approach to constraint LTL, Information and Computation 205 (3) (2007) 380–415.

[29] Rajeev Alur, David L. Dill, A theory of timed automata, Theoretical Computer Science 126 (1994) 183?235.

[30] M. Quartulli, J. Lozano, 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.

[31] [link].
URL https://github.com/DigitalPebble

[32] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2001.

[33] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: an experience report on migration to a cloud-native architecture.

[34] S. Peng, J. Gu, X. S. Wang, W. Rao, M. Yang, Y. Cao, Cost-based optimization of logical partitions for a query workload in a hadoop data warehouse., in: L. Chen, Y. Jia, T. K. Sellis, G. Liu (Eds.), APWeb, Vol. 8709 of Lecture Notes in Computer Science, Springer, 2014, pp. 559–567.
URL http://dblp.uni-trier.de/db/conf/apweb/apweb2014.html#PengGWRYC14

[35] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. P. Mendell, H. Nasgaard, S. Schneider, R. Soul, K.-L. Wu, Ibm streams processing language: Analyzing big data in motion., IBM Journal of Research and Development 57 (3/4) (2013) 7.

[36] D. Wang, J. Liu, Optimizing big data processing performance in the public cloud: opportunities and approaches., IEEE Network 29 (5) (2015) 31–35.

[37] Y. Tamura, S. Yamada, Reliability analysis based on a jump diffusion model with two wiener processes for cloud computing with big data., Entropy 17 (7) (2015) 4533–4546.

[38] K. Chandrasekaran, S. Santurkar, A. Arora, Stormgen - a domain specific language to create ad-hoc storm topologies., in: FedCSIS, 2014, pp. 1621–1628.

[39] 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, ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 638–641. doi:10.1145/2591062.2591088.

[40] M. M. Tommaso Di Noia, E. D. Sciascio, A computational model for mapreduce job flow (2014).

[41] F. Yang, W. Su, H. Zhu, Q. Li, Formalizing mapreduce with csp, in: Proceedings of ECBS, IEEE Computer Society, 2010, pp. 358–367. doi:10.1109/ECBS.2010.50.

[42] [link].
URL https://github.com/maelstromdat/OSTIA