Verifying Big Data Topologies *By-Design*: a Semi-Automated Approach

Marcello M. Bersani · Francesco Marconi · Damian A. Tamburri · Andrea Nodari · Pooyan Jamshidi

Abstract Big data architectures have been gaining momentum in recent years. For instance, Twitter uses stream processing frameworks like Apache 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 (Ordinary Static Topology Inference Analysis) that allows detecting the occurrence of common anti-patterns across big data architectures and 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.

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 [19,30]. These applications are receiving

Marcello and Francesco
Politecnico di Milano, Milan, Italy
E-mail: [marcellomaria.bersani, francesco.marconi]@polimi.it

Damian
TU/e - JADS
E-mail: d.a.tamburri@tue.nl

Pooyan and Andrea
Imperial College London
E-mail: [p.jamshidi,a.nodari15]@imperial.co.uk

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 infrastructure costs, steep learning curves for the different frameworks (such as Apache Storm [20], Apache Spark [2] or Apache Hadoop [3]) 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 [12]. The process in question is especially complex as the number of middleware involved in DIAs design increases; the more middleware are involved the more parameters need co–evaluation (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.*

We argue that a primary entry-point for such fine-tuning is the DIA's graph of operations along with the configurations that the graph is decorated with, for execution. On one hand, the graph in question is a DAG — a Directed Acyclic Graph representing 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. On the other hand, the topology can either be known to the designer or it can be directly extracted from DIA code. This second scenario is where our research solution comes in.

This paper illustrates and evaluates OSTIA, which stands for "Ordinary Static Topology Inference Analysis" – OSTIA is a tool which retrieves data-intensive topologies to allow for: (a) *anti-pattern analysis* - OSTIA allows detection of known and established design anti-patterns for data-intensive applications; (b) *transparent formal verification* - OSTIA transposes the recovered data-intensive topology models into equivalent formal models for the purpose of verifying temporal properties, such as basic queue-safety clauses [11].

First, during its reverse-engineering step, OSTIA recovers a JSON file describing the technical structure details and configurations in the targeted topologies.

Secondly, such representations may be used for further analysis through model verification in a completely transparent fashion thanks to formal verification techniques [11].

This paper outlines OSTIA, elaborating on the major usage scenario above, its 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

Table 1 Focus-Groups population outline.

Role	#Participants	Mean Age	Mean Exp. With DIAs (#months)
Architect	3	35,3	17,3
Developer	4	27,7	36,2
Operator	5	31,1	38,1
Manager	3	44,2	18,4

previous version of this paper was published in the proceedings of WICSA 2015 [12], we introduce the following novel contributions:

– we extended OSTIA to address Apache Hadoop data-intensive applications and re-executed the evaluation in line with this addition;
– we extended OSTIA with a formal verification feature for using a formal model built via CLTLoc - this feature operates LTL Constraint verification over-clocks and is completely transparent to OSTIA users, checking autonomously for safety of OSTIA-elicited topologies;

The rest of the paper is structured as follows. The next section elaborates further on the emerging notion of continuous architecting for DIAs. Section 2 outlines our research design and context of study. Section 3 outlines OSTIA. Section 4 evaluates OSTIA while Section 5 discusses results and evaluation outlining OSTIA limitations and threats to validity. Finally, Sections 6 and 7 report related work and conclude the paper.

2 Research Methods

From a methodological perspective, the results outlined in this paper were elaborated as follows.

2.1 Extracting Anti-Patterns for Big Data Applications

he anti-patterns illustrated in this paper were initially elaborated within 3 structured focus-groups [25] involving practitioners from a different organization in each focus-group round; subsequently, we interviewed 2 domain-expert (5+ years of experience) researchers on big data technologies as a control group. The data was analyzed with a simple card-sorting exercise and cross-referenced the patterns thus emerged with the ones emerging from our interview-based control group; disagreement between the two groups was evaluated Inter-Rater Reliability assessment using the well-known Krippendorf Alpha coefficient [23] (assessment of $K_{alpha} = 0,89$).

Table 1 outlines the population we used for this part of the study. The practitioners were simply required to elaborate on the most frequent structural and anti-patterns they encountered on their DIA design and experimentation.

The focus-group sessions were structured as follows: (a) the practitioners were presented with a data-intensive architectural design using standard UML

structure and behavior representations (a component view and an activity view [**?**]); (b) the practitioners were asked to identify and discuss any bottlenecks or structural limitations in the outlined designs; (c) finally, the practitioners were asked to illustrate any other anti-pattern the showcased topologies did not contain.

2.2 Research Solution Evaluation

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. 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" [6]; (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" [7].

Third, finally, as part of the OSTIA extension recapped in this manuscript, we applied formal verification approaches using the Zot [21] LTL model-checker following an approach tailored from previous work [11,13].

3 OSTIA Explained

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 design-time analysis and continuous improvement of data-intensive applications, 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.

3.1 OSTIA Tool Architecture

The overall architecture of OSTIA is depicted in Figure 1. 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 additional algorithmic analyses similar to our anti-pattern analyses can be easily added. These analyses use the information resides in the intermediate format and provide added value analyses for design-time analysis and verification. 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.
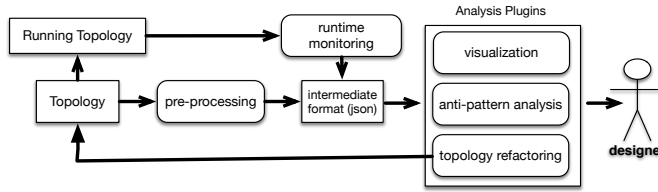

Fig. 1 OSTIA extensible architecture.

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 (specifically, JSON, Dot, CSV, and XMI) to analyse and continuously improve the topology with other tools — in the scope of this paper we focus on verification *by-design* featuring formal verification.

### 3.2 Topology Design Anti-Patterns Within OSTIA

This section elaborates on the anti-patterns we elicited (See Section 2). 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.

3.2.1 Multi-Anchoring

The Multi-Anchoring pattern is shown in Fig. 2. In order to guarantee fault-tolerant stream processing, tuples processed by bolts need to be anchored with
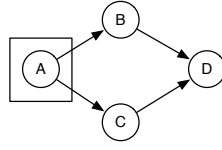
(a) Multi-anchoring.

Fig. 2 The multi-anchoring anti-pattern.
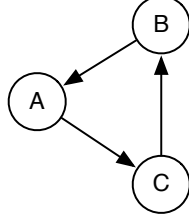
(a) Cycle-in.

Fig. 3 The cycle-in anti-pattern.

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. Our practitioners agree that multiple ackers can indeed cause much overhead and influence the operational performance of the entire topology.

3.2.2 Cycle-in Topology

The Cycle-in pattern is shown in Fig. 3. 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 which in turn incurs in increased costs.

3.2.3 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.
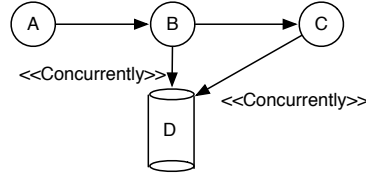
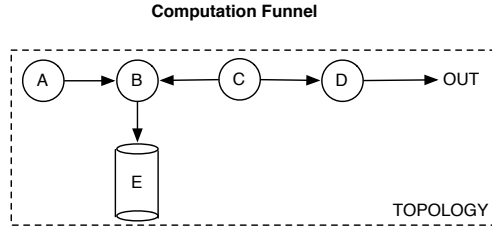**Fig. 4** Concurrency management in case of Persistent Data circumstances.

Fig. 5 computation funnel.

3.2.4 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.

3.3 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 [24], 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) [18] 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:

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

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 [18,13] 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 [13]. 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 [24] 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;
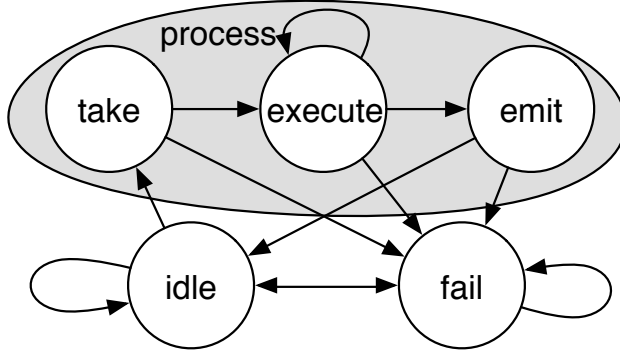
Fig. 6 Finite state automaton describing bolt states.

– 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 jth 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 [24] is shown in Fig. 6. We provide, as an example, one of the formulae defining the processing state. Formula 1 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"*.

$$\bigwedge_{i \in \mathbf{B}} \begin{pmatrix} \mathtt{process}_i \Rightarrow \\ \mathtt{process}_i \, \mathbf{S} \, (\mathtt{take}_i \vee (\mathtt{orig} \wedge \mathtt{process}_i)) \wedge \\ \mathtt{process}_i \, \mathbf{U} \, (\mathtt{emit}_i \vee \mathtt{fail}_i) \wedge \neg \mathtt{fail}_i \end{pmatrix} \tag{1}$$

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 2, 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"*.

$$\bigwedge_{j \in B} \left(\texttt{take}_j \Rightarrow (\mathrm{X}q_j = q_j + r_{\texttt{add}_j} - r_{\texttt{take}_j}) \right) \tag{2}$$

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.

4 Evaluation

We evaluated OSTIA through qualitative evaluation and case-study research featuring an open-/closed-source industrial case study (see Section 4.1) and two open-source case studies (see Section 4.2) on which we also applied complex formal verification (see Section 3.3).

4.1 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 architecture refactoring 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 helping designers in tuning their application through a design-and-refactor loop.

In addition to formal verification, specific algorithms for graph analysis can be integrated in OSTIA to offer a deeper insight of the applications. For instance, the industrial case study has been analyzed with two algorithms to identify linear sequences of nodes and clusters in the topology graph. Topology linearisation results in sorting the processing elements in a topology in a
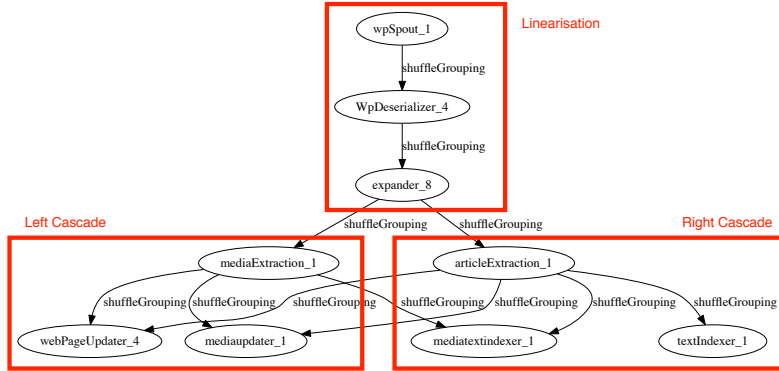
Fig. 7 SocialSensor App, OSTIA sample output partially linearised (top) and cascaded (bottom left and right).

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. 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. Clusters may require, in general, 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.

OSTIA standard output[1] for the smallest of the three SocialSensor topologies, namely the "focused-crawler" topology, is outlined in Fig. 7.

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

---

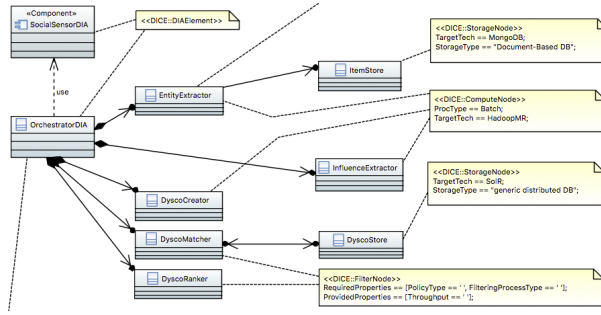[1] Output of OSTIA analyses is not evidenced for the sake of space.

Fig. 8 Industrial case-study, a refactored architecture.

the refactoring of the topology in question as part of the needed architectural reasoning.

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.

As a followup of our analysis, our partner is refactoring his own high-level software architecture adopting a lambda-like software architecture style [28] (see Fig. 8) which includes the Social-Sensor App (Top of Fig. 8) 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. 8) and intents but batched out of the topological processing in Storm (see Fig. 8)[2].

Our qualitative evaluation of the refactored architecture by means of several interviews and workshops revealed very encouraging results.

4.2 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 [8] and StormCV [7] applications. Figures 10 and 9 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.

---

[2] several other overburdened topological elements were refactored but were omitted here due to industrial secrecy
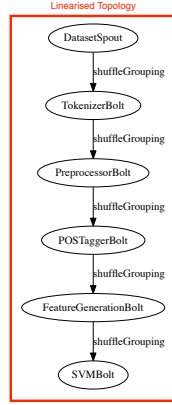
Fig. 9 StormCV topology (linearised).

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. 10 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.

4.3 OSTIA-based 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 improving these topologies through refactoring.

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 11 shows how the tool constructed a periodic model in which a suffix (highlighted by the gray background) of a finite
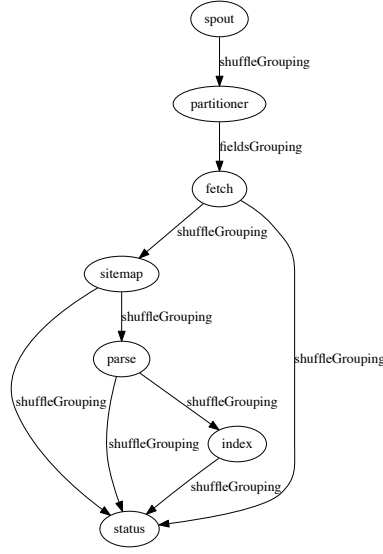
Fig. 10 DigitalPebble topology.

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. 11, 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 models of the topologies making almost invisible the formal verification layer (other than manually setting and tuning operational parameters for verification).

5 Discussion

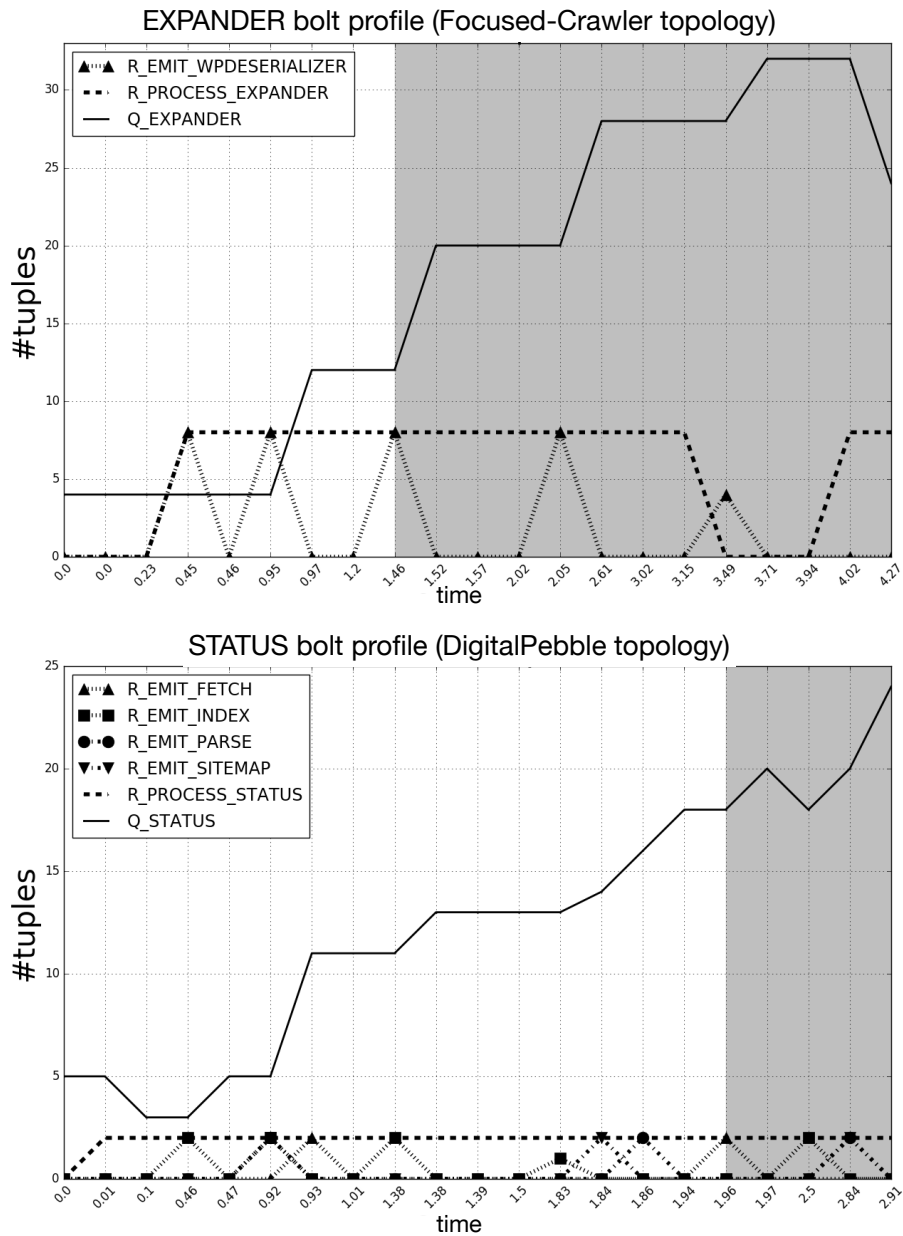This section discusses some findings and the limitations of OSTIA.

Fig. 11 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).

5.1 Findings and Observations

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 [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 [28] and Microservices [10]. OSTIA may aid in this investigation by allowing the interactive and incremental improvement of multiple (combinations of) topologies together.

5.2 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. [26] and similar.

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.

6 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 [22] 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 [33] or reliability point of view [31]. 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 [32], or the validity of the messaging flow in MapReduce [34].

7 Conclusion

We set out to assist the design-time formal verification of big data 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 [9].

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 com-

mon application framework for streaming, i.e., Storm. Finally, we plan to further evaluate OSTIA using empirical evaluation.

BibTeX users please use one of

References

1. URL `http://www.gartner.com/newsroom/id/2637615`
2. URL `http://spark.apache.org/`
3. URL `https://hadoop.apache.org/`
4. URL `http://www.dice-h2020.eu/`
5. URL `https://github.com/socialsensor`
6. URL `https://github.com/DigitalPebble/storm-crawler`
7. URL `https://github.com/sensorstorm/StormCV`
8. URL `https://github.com/DigitalPebble`
9. URL `https://github.com/maelstromdat/OSTIA`
10. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: an experience report on migration to a cloud-native architecture (2016)
11. Bersani, M.M., Distefano, S., Ferrucci, L., Mazzara, M.: A timed semantics of workflows. In: ICSOFT (Selected Papers), *Communications in Computer and Information Science*, vol. 555, pp. 365–383. Springer (2014)
12. Bersani, M.M., Marconi, F., Tamburri, D.A., Jamshidi, P., Nodari, A.: Continuous Architecting of Stream-Based Systems. In: H. Muccini, E.K. Harper (eds.) Proceedings of the 25th IFIP / IEEE Working Conference on Software Architectures, pp. 131–142. IEEE Computer Society (2016)
13. Bersani, M.M., Rossi, M., San Pietro, P.: A tool for deciding the satisfiability of continuous-time metric temporal logic. Acta Informatica pp. 1–36 (2015). DOI 10.1007/s00236-015-0229-y
14. Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., Herbst, N., Jamshidi, P., Jung, R., von Kistowski, J., et al.: Performance-oriented devops: A research agenda. arXiv preprint arXiv:1508.04752 (2015)
15. Camilli, M.: 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, pp. 638–641. ACM, New York, NY, USA (2014). DOI 10.1145/2591062.2591088
16. Chandrasekaran, K., Santurkar, S., Arora, A.: Stormgen - a domain specific language to create ad-hoc storm topologies. In: FedCSIS, pp. 1621–1628 (2014)
17. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley (2001)
18. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Information and Computation **205**(3), 380–415 (2007)
19. Emani, C.K., Cullot, N., Nicolle, C.: Understandable big data: A survey. Computer Science Review **17**, 70–81 (2015)
20. Evans, R.: Apache storm, a hands on tutorial. In: IC2E, p. 2. IEEE (2015)
21. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling time in computing: A taxonomy and a comparative survey. ACM Comput. Surv. **42**(2), 6:1–6:59 (2010)
22. Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M.P., Nasgaard, H., Schneider, S., Soul, R., Wu, K.L.: Ibm streams processing language: Analyzing big data in motion. IBM Journal of Research and Development **57**(3/4), 7 (2013)
23. Krippendorff, K.: Content Analysis: An Introduction to Its Methodology (second edition). Sage Publications (2004)

24. Marconi, F., Bersani, M.M., Erascu, M., Rossi, M.: Towards the formal verification of dia through mtl models. In: booktitle, Lecture Notes in Computer Science
25. Morgan, D.L.: Focus groups as qualitative research. Sage Publications, Thousand Oaks (1997)
26. Peng, S., Gu, J., Wang, X.S., Rao, W., Yang, M., Cao, Y.: 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, *Lecture Notes in Computer Science*, vol. 8709, pp. 559–567. Springer (2014). URL `http://dblp.uni-trier.de/db/conf/apweb/apweb2014.html#PengGWRYC14`
27. Pradella, M., Morzenti, A., Pietro, P.S.: Bounded satisfiability checking of metric temporal logic specifications. ACM Trans. Softw. Eng. Methodol. **22**(3), 20:1–20:54 (2013). DOI 10.1145/2491509.2491514
28. Quartulli, M., Lozano, J., Olaizola, I.G.: Beyond the lambda architecture: Effective scheduling for large scale eo information mining and interactive thematic mapping. In: IGARSS, pp. 1492–1495 (2015)
29. Rajeev Alur, David L. Dill: A theory of timed automata. Theoretical Computer Science **126**, 183?235 (1994)
30. Ratner, B.: Statistical and Machine-Learning Data Mining: Techniques for Better Predictive Modeling and Analysis of Big Data. CRC PressINC (2012)
31. Tamura, Y., Yamada, S.: Reliability analysis based on a jump diffusion model with two wiener processes for cloud computing with big data. Entropy **17**(7), 4533–4546 (2015)
32. Tommaso Di Noia, M.M., Sciascio, E.D.: A computational model for mapreduce job flow (2014)
33. Wang, D., Liu, J.: Optimizing big data processing performance in the public cloud: opportunities and approaches. IEEE Network **29**(5), 31–35 (2015)
34. Yang, F., Su, W., Zhu, H., Li, Q.: Formalizing mapreduce with csp. In: Proceedings of ECBS, pp. 358–367. IEEE Computer Society (2010). DOI 10.1109/ECBS.2010.50