Bersani *et al. J Big Data* (2019) 6:40
https://doi.org/10.1186/s40537-019-0199-y

Journal of Big Data

METHODOLOGY Open Access

Verifying big data topologies *by-design*: a semi-automated approach

Marcello M. Bersani[1*], Francesco Marconi[1], Damian A. Tamburri[2*], Andrea Nodari[3] and Pooyan Jamshidi[3]

*Correspondence: marcellomaria.bersani@polimi.it; d.a.tamburri@tue.nl
[1] Politecnico di Milano, Milan, Italy
[2] TU/e - JADS, Eindhoven, The Netherlands
Full list of author information is available at the end of the article

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. Such complex architectures make possible refactoring of the applications a difficult task for software architects, as applications might be very different with respect to 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.

Keywords: Big data architectures, Software design and analysis, Big data systems verification

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 [20, 35]. These applications are receiving increased attention in the last years given their ability to yield competitive advantage by direct investigation of user needs and trends hidden in the enormous quantities of data produced daily by the average Internet user. According to Gartner [1] business intelligence and analytics applications will remain a top focus for Chief-Information Officers (CIOs) of most Fortune 500 companies until at least 2019–2021. 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 [21], Apache Spark [2] or Apache Hadoop [3]) typically involved in design and development of big data applications and complexities in large-scale architectures.

A key complexity of the above design and development activity lies in quickly and continuously refining the configuration parameters of the middleware and service platforms on top of 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. This is possible when the adopted framework decomposes the computation in term of concurrent operations on data that are subject to a specific precedence relation. 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 application graph 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 thanks to formal verification techniques [11]. The verification approach is lightweight and it is carried out in a completely transparent fashion to OSTIA users.

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 refactoring of big data architectures. Although a 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 Constraint LTL over-clocks (CLTLoc) [13]—an extension of the well-known Linear Temporal Logic (LTL) [31] with variables measuring the elapsing of time. This feature operates verification on CLTLoc specifications and is completely transparent to OSTIA users, checking autonomously for safety of OSTIA-elicited topologies;

We released OSTIA as an open-source software [4].

The rest of the paper is structured as follows. The next section elaborates further on the notion of refactoring for DIAs. "Research methods" section outlines our research design and context of study. "Results: OSTIA explained" section outlines OSTIA. "Results" section evaluates OSTIA while "Discussion" section discusses results and evaluation outlining OSTIA limitations and threats to validity. Finally, "Related work" and "Conclusion" sections report related work and conclude the paper.

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

Research methods

From a methodological perspective, the results outlined in this paper were elaborated as follows and made concrete through the actions in "Extracting anti-patterns for big data applications" and "Research solution evaluation" sections.

Extracting anti-patterns for big data applications

The anti-patterns illustrated in this paper were initially elaborated within three structured focus-groups [28] involving practitioners from a different organization in each focus-group round; subsequently, we interviewed two 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. The patterns emerged from the card-sorting were confirmed/disproved with the patterns emerging from our interview-based control group; disagreement between the two groups was evaluated Inter-Rater Reliability assessment using the well-known Krippendorff Alpha coefficient [26] (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 [19]); (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.

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 [5]. 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 [6] 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 and can be used as a front end to various ML algorithms" [7]; (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" [8].

Third, finally, as part of the OSTIA extension recapped in this manuscript, we applied formal verification approaches using the Zot [23] model-checker following an approach tailored from previous work [11, 13].

Results: OSTIA explained

This section introduces how OSTIA was designed to support design-time analysis and continuous improvement of data-intensive applications, using the Storm framework as a running example. For this reason, a brief recap of Storm is given to understand the rationale behind OSTIA.

A concrete example: the storm architecture

Storm is a technology developed at Twitter [39] 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 DAG composed by nodes of two types: spouts and bolts. The former type includes nodes that process the data entering the topology, for instance querying APIs or retrieve information from a message broker, such as Apache Kafka.[1] The latter executes operations on data, such as filtering or serialising.

OSTIA tool architecture

Architecture overview

The overall architecture of OSTIA is depicted in Fig. 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 afterwards by other algorithmic processes.

OSTIA is indeed architected in a way that additional algorithmic analyses similar to our anti-pattern analyses can be easily added. These functionalities are carried out with the information that resides in the intermediate format and provide added value for the design-time analysis and verification. Since the information in the intermediate format only rely on the logical code analysis, the algorithmic analyses require some additional information regarding the running topology, such as, for instance, the end to end latency

[1] http://kafka.apache.org/.

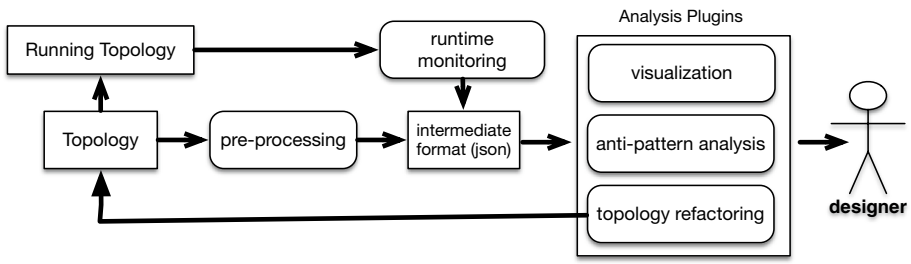

Fig. 1 OSTIA extensible architecture

and throughput of the topology or the mean duration of the computation carried out by the computational nodes when they process a unit of data.

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.

Architecture properties and extensibility

The architectural design of the OSTIA tool was incepted using a modular model-driven architecture [22] in mind. More specifically, the tool provides a platform-independent and topology-based analysis module which elicits topologies from data-intensive applications using an technology-agnostic format based on the ".Dot" notation, a well-known standard graph-representation format. On top of this analysis module, the architecture provides a design and analysis module which outputs a visualization of the graph-formatted input. Finally, the tool provides a pattern-analysis module with graph-analysis and pattern-mining functions; one function per pattern is used in this module. Finally, the tool provides a software-verification interlay relying on third-party tools from previous and related work as outlined in "OSTIA-based formal verification" section.

From an extensibility perspective, the architecture provides a basis template commented within the source-code as a basic format to be used to extend each module; in principle, extending designers need to simply "instantiate" this template within the module and recall the extension from the visualization layer to warrant for OSTIA extensibility.

OSTIA methodology

The OSTIA Methodology effectively combines two successful approaches commonly adopted software development. The first one is DevOps and the second one is Model-Driven Engineering. OSTIA can be adopted by both the Developers and Operators parts of the DevOps cycle that, together, contribute to the iterative developments cycle of software; and, in addition, it can be used to effectively enforce the model

refinement that enables the shift from high-level abstract models to low-level refined ones.

OSTIA takes part in the design process at the level of Developers as follows. Designers of applications can use OSTIA to model their application by means of an abstract modeling language, based on UML. The language allows them to design the application in terms of abstraction that model the computational nodes of the application and the data sources providing input data. Based on the adopted technology, that will be used for the implementation of the final artifact, the language offers suitable stereotypes modeling the relevant technology-dependent features and that enable the analysis of the application design by means of the OSTIA verification tool. This work focuses on two specific technologies and, therefore, the UML abstractions are only limited to those required to model Apache Storm applications and Hadoop applications. Moreover, on the Developers side, the designers can use OSTIA to iteratively refine the model of their application by running the automatic analysis on different application models, that are possibly instantiated with different parameter values (e.g., the number of workers in a node running a certain functionality of the Storm topology).

On the other hand, OSTIA also participates to the DevOps cycle in the Operators side because it offers post-design analysis features. OSTIA, in fact, can be adopted by operators for the elicitation of the application architecture from its source code. In particular, a number of structural anti-pattern has been identified in this work as potential threats that might affect the performance of the application and even its correct behavior at runtime. OSTIA implements basic yet useful functionalities for static code analysis that can be used by designers and operators to discover possibly structural issues. The result of the analysis that OSTIA provides at this level is the application topology and the parts of the application that are likely to be a potential threat for the entire application. Combining the application topology with runtime information, that can be collected by standard monitoring framework, the designers can actually enforce a refinement iteration on their design, in addition to the one performed at design time, that is based on realistic information coming from the real deployment of the application. This step might turn out in a refactoring of the deployed design into a new refined solution that, in turn, can be verified with the OSTIA verification tool, deployed and later analyzed with the same OSTIA functionalities. Figure 2 shows the refinement loop which is enabled by OSTIA.

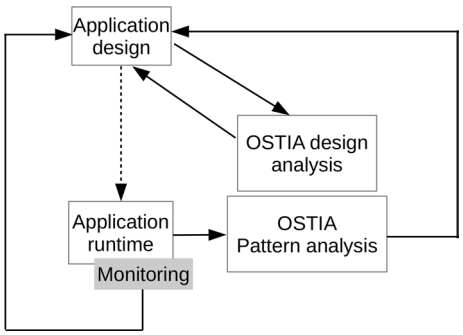

Fig. 2 Iterative refinement support by OSTIA

To make the OSTIA methodology a practice, the following activities reflected into the OSTIA tool.

- *Architecture elicitation* The static analysis of the source code of the application extracts its topology and made it available for later analysis.
- *Structural anti-pattern identification* Standard algorithms for graph analysis (such as clustering) identify specific structures in the application topology that might lead to undesired behaviors.
- *Formal analysis* Model-checking of the annotated model of the application verifies the existence of executions that might burden the application runtime with an excessive workload.

The previous tools can be used in the following scenarios.

- *Architecture analysis* A development team implements an application that has to satisfy certain requirements at runtime. OSTIA can be used to refine the application model before the implementation phase.
- *DevOps* As part of a DevOps pipeline dedicated to data-intensive solutions, OSTIA can be used for instrumenting the continuous refactoring of the data-intensive application by studying the application structure and the underlying topology to improve their operational characteristics.

Topology design anti-patterns within OSTIA

This section elaborates on the anti-patterns we elicited (see "Research methods" section). 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.

Multi-anchoring

The multi-anchoring pattern is shown in Fig. 3. 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. Our practitioners agree that

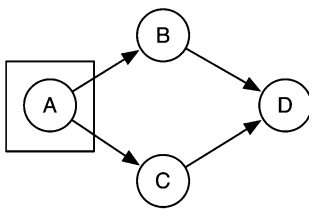

Fig. 3 The multi-anchoring anti-pattern

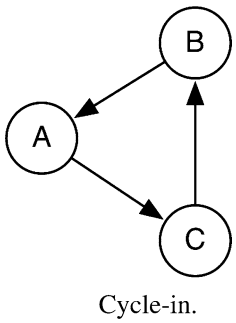

Fig. 4 The cycle-in anti-pattern

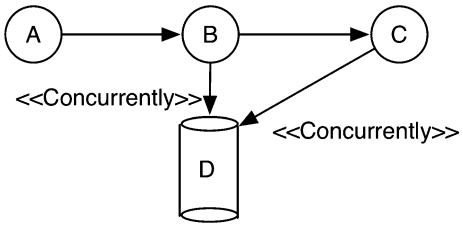

Fig. 5 Concurrency management in case of persistent data circumstances

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

Cycle-in topology

The cycle-in pattern is shown in Fig. 4. 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.

Persistent data

The persistent data pattern is shown in Fig. 5. 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.

Computation funnel

The computational funnel is shown in Fig. 6. 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

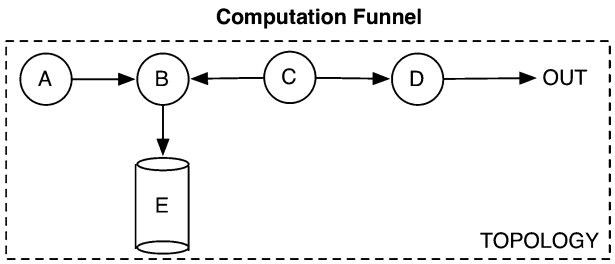

Fig. 6 Computation funnel

circumstance should be dealt with since it may compromise the availability of results under the desired performance restrictions.

DOT format for topology elicitation

As previously stated, the OSTIA tool is rigged to elicit and represent Big Data topologies using the "*.dot" format; the format in question is a de-facto and de-iure graph description language. DOT graphs are typically files with the file extension gv or dot. Paraphrasing from Wikipedia, *"Various programs can process DOT files. Some, such as dot, neato, twopi, circo, fdp, and sfdp, can read a DOT file and render it in graphical form. Others, such as gvpr, gc, acyclic, ccomps, sccmap, and tred, read DOT files and perform calculations on the represented graph. Finally, others, such as lefty, dotty, and grappa, provide an interactive interface [...]"*. A small excerpt of DOT code describing a graph with 4 nodes is the following:

Listing 1 DOT script describing an undirected graph N with four nodes.

```
graph N {
  n1 -- n2 -- n3;
  n2 -- n4;
}
```

OSTIA uses the same approach as the aforementioned tools and instatiates the same design-patterns employed by the tools in question to enact formal-verification of data-intensive topologies.

OSTIA-based formal verification

This section describes the formal modelling and verification employed in OSTIA. Our assumption for DIA refactoring 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 [27], relies on *satisfiability checking* [32], 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) [34]. 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:

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

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 $\mathrm{Xy} \sim \mathrm{z} \pm \mathrm{c}$. Intuitively, formula $x \sim c$ states that the value of clock x is $\sim$ than/to c and formula $\mathrm{Xy} \sim \mathrm{z} \pm \mathrm{c}$ 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 [13, 18] but, for practical implementation, Bounded Satisfiability Checking (BSC) [32] 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 [27] 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;

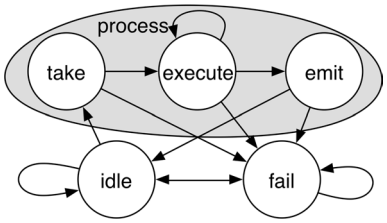

Fig. 7 Finite state automaton describing bolt states

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 [27] is shown in Fig. 7. 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".*

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

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".*

$$\bigwedge_{j \in B} (\texttt{take}_j \Rightarrow (\mathrm{X} \mathbf{q_j} = \mathbf{q_j} + \mathbf{r}_{\texttt{add}_j} - \mathbf{r}_{\texttt{take}_j})) \tag{2}$$

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.

JSON format for verification

Listing 3.7 shows an excerpt of a JSON script describing a topology including two spouts, called S_1 and S_2, and three bolts, called called B_1, S_2 and S_3. Spouts and bolts are modeled by means of a number of parameters that represent an abstraction of their (non-functional) behavior at runtime. The JSON format is a readable means that captures all the needed information, required to run the verification, that are classified into three distinct groups. A list of the main ones is included hereafter.

- Topology-related settings:
  - list of spouts:
    - `emit_rate`: spout average tuple emitting rate.
  - list of bolts:
    - `subs`: the list of all the nodes in the topology that send tuple to the bolt.
    - `parallelism`: level of parallelism chosen for the bolt. This value can be extracted from the code implementing Storm topology or set at design time.
    - `alpha`: average processing time for the single tuple.
    - `sigma`: ration between number of output tuples and number of input tuples. This value is an abstraction of the functionality carried out by the bolt: values smaller than one model filtering functions whereas value greater than one model other generic function on input tuples.
  - structure of the topology, expressed through the combination of the subscription lists ("`subs`") of all the bolts composing the topology.
  - `queue_threshold`: the maximum level of occupancy that should not be exceeded by any queue. This value is extracted from the code implementing Storm topology or set at design time.
  - `max_idle_time`: the maximum time for a bolt to be inactive.
- Verification-related settings: the information in this section does not model the topology itself but actually relates to the analysis that is run on the topology.
  - `num_steps`: being the verification engine implemented according to the bounded model-checking approach, the value specifies the number of discrete time instants to be explored in the verification phase.
  - `periodic_queues`: the list of bolts whose queue size is analyzed. The verification procedure determines the existence of a system execution that leads to and increasing queue size for the bolts specified in the list.
  - `plugin`: underlying model-checker to be used.

Listing 2 JSON script describing a simple topology. Dots are used as abbreviations.

```
{
 "app_name": "Simple Topology",
 "description": "",
 "version": "0.1",
 "topology":{
    "spouts":[
                {"id":"S1",
                "avg_emit_rate":2.0},
                {"id":"S2",
                "avg_emit_rate":1.0}
             ],
    "bolts":[
                {"id": "B1",
                ...},
                {"id": "B2",
                "subs": ["S1", "S2"],
                "alpha": 5.0,
                "sigma": 0.5,
                "min_ttf": 1000,
                "parallelism": 10},
                {"id": "B3",
                ...}
            ],
    "max_idle_time": 0.01,
    "queue_threshold": 2000
 },
 "verification_params":{
            "plugin" : ...,
            "max_time" :  ...,
            "num_steps": ...,
            "periodic_queues":["B1"]}
 }
```

Results

We evaluated OSTIA through qualitative evaluation and case-study research featuring an open-/closed-source industrial case study (see "Establishing anti-patterns occurrence with case-study research: 3 cases from industry" section) and two open-source case studies (see "Establishing anti-patterns occurrence with case-study research: 3 cases from open-source" section) on which we also applied OSTIA-based formal verification and refactoring (see "OSTIA-based formal verification" section). The objective of the evaluation was twofold:

OBJ.1 Evaluate the occurrence of anti-patterns evidenced by our practitioners in both open- and closed-source DIAs;

OBJ.2 Understand whether OSTIA-based analyses aid in refactoring towards formally-verified DIA topologies *by-design*;

Establishing anti-patterns occurrence with case-study research: 3 cases from industry

OSTIA was evaluated using 3 medium/large topologies (11+ elements) part of the SocialSensor App. Our industrial partner is having performance and availability outages connected to currently unknown circumstances. Therefore, the objective of our

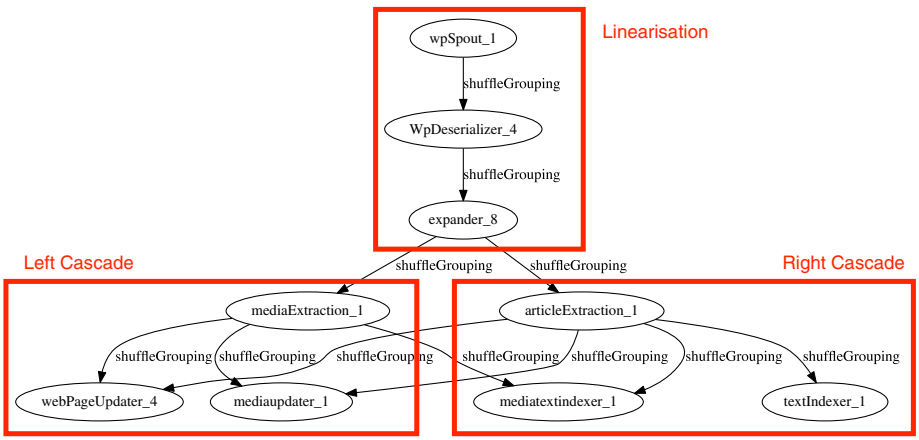

Fig. 8 SocialSensor App, OSTIA sample output partially linearised (top) and cascaded (bottom left and right)

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 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[2] for the smallest of the three SocialSensor topologies, namely the "focused-crawler" topology, is outlined in Fig. 8.

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

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

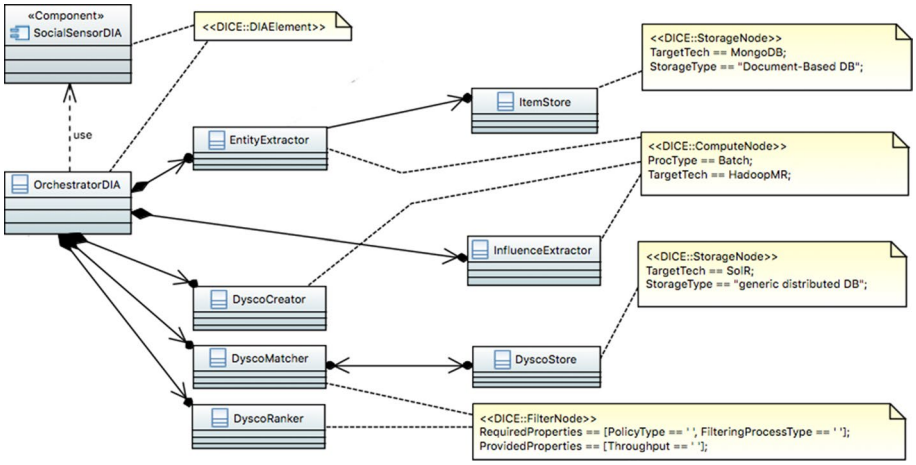

Fig. 9 Industrial case-study, a refactored architecture

OSTIA 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 [33] (see Fig. 9) which includes the Social-Sensor App (Top of Fig. 9) 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 within the Social Sensor app to additional ad-hoc Hadoop Map Reduce jobs with similar purpose (e.g., the EntityExtractor compute node in Fig. 9) and intents but batched out of the topological processing in Storm (see Fig. 9).[3]

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

Establishing anti-patterns occurrence with case-study research: 3 cases from open-source

To confirm the usefulness and capacity of OSTIA to enact a refactoring cycle, we applied it in understanding (first) and attempting improvements of two open-source applications, namely, the previously introduced DigitalPebble [9] and StormCV [8] applications. Figures 10 and 11 outline standard OSTIA output for the two applications. Note that we did not have any prior knowledge concerning the two applications in question and we merely run OSTIA on the applications' codebase dump in our own experimental machine. OSTIA output takes mere seconds for small to medium-sized topologies (e.g., around 25 nodes).

The OSTIA output aided as follows: (a) the output summarised in Fig. 11 allowed us to immediately grasp the functional behavior of the DigitalPebble and StormCV

[3] Several other overburdened topological elements were refactored but were omitted here due to industrial secrecy.

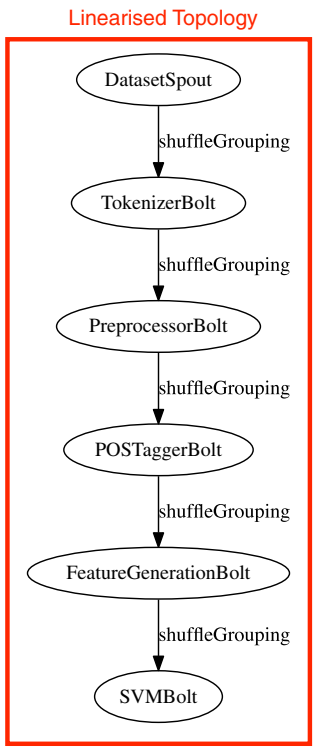

Fig. 10 StormCV topology (linearised)

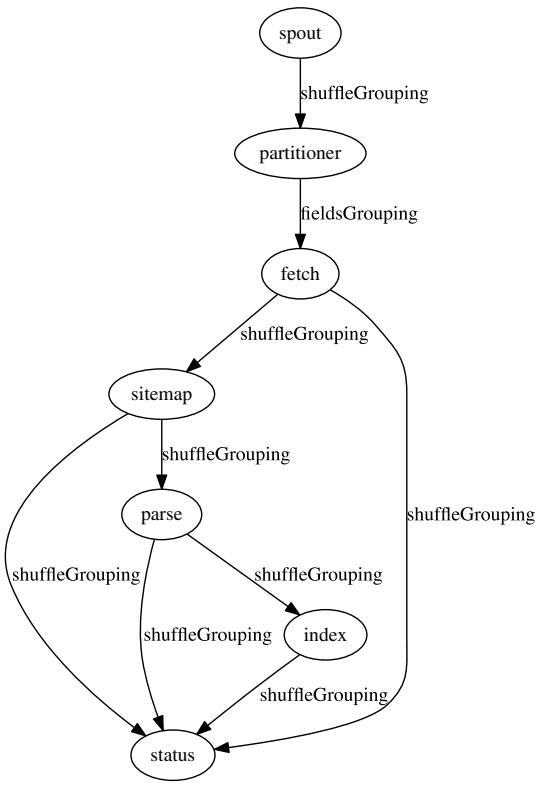

Fig. 11 DigitalPebble topology

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.

Summary for Obj 1. The patterns we elicited thanks to focus-groups in industry indeed have an actual recurrent manifestation in both industry and open-source. OSTIA-based analysis can support reasoning and potential refactoring of the proposed anti-patterns.

OSTIA-based formal verification and refactoring

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 12 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. 12, 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).

Summary for Obj 2. OSTIA-based formal verification effectively evaluates the safety of DIAs focusing on their design-time representation; further investigation of the generalisability of this approach towards runtime is needed to scope the extent to which OSTIA offers support for continuous evolution.

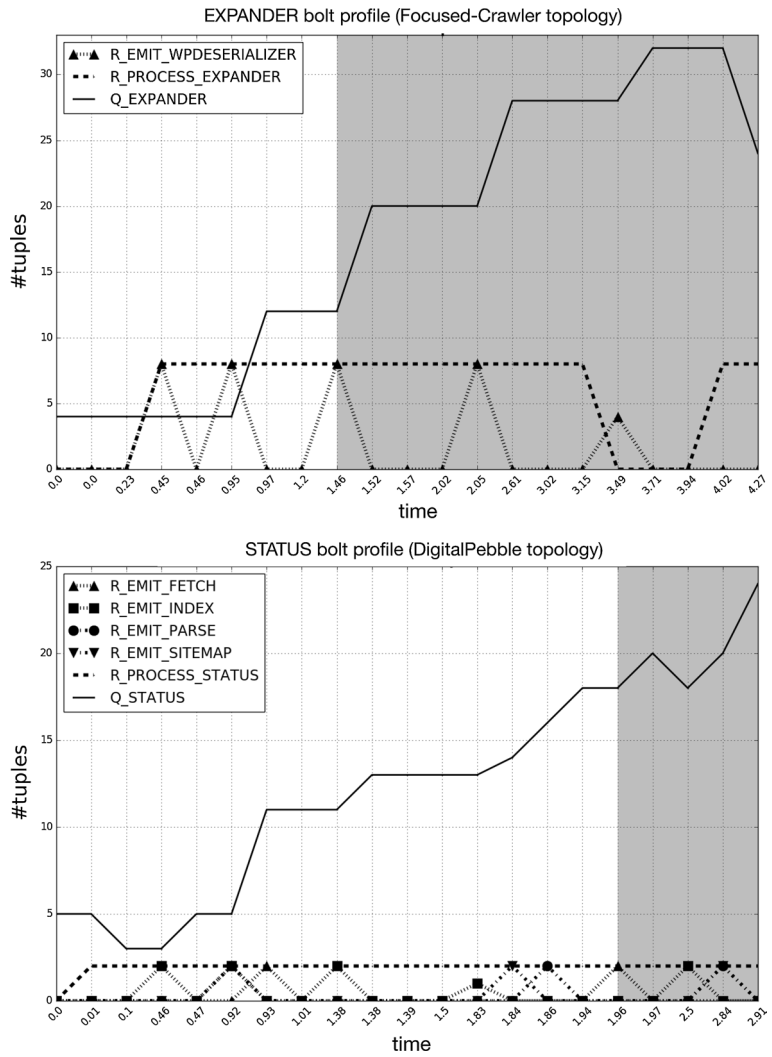

Fig. 12 OSTIA-based formal verification output traces showing the evolution of the two bolts over time. Queue trends are displayed as solid black line. Dashed lines show the processing activity of the bolts, while the other lines illustrate the incoming tuples from the subscribed nodes (`emit` events)

Discussion

This section discusses some findings and the limitations of OSTIA.

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 application refactoring.

First, we found (and observed in industrial practice) that it is often common to develop "runnable" architecture topology that will undergo for refactoring even after the deployment phase and while the application is running. 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 the refactoring can help in boosting the application, that therefore require 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 [33] and Microservices [10]. OSTIA may aid in this investigation by allowing the interactive and incremental improvement of multiple (combinations of) topologies together.

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. [30] 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 the refactoring process.

Related work

The work behind OSTIA stems from the EU H2020 Project called DICE [5] where we are investigating the use of model-driven facilities to support the design and quality enhancement of big data applications. Much similarly to the DICE effort, the IBM Stream Processing Language (SPL) initiative [24] provides an implementation language specific to programming streams management (e.g., Storm jobs) and related reactive systems. In addition, there are several work close to OSTIA in terms of their foundations and type of support, e.g., works focusing on distilling and analysing big data topologies *by-design* [36], as also highlighted in recent research by Kalantari et al. [25].

First, from a non-functional perspective, much literature discusses quality analyses of Big Data topologies, e.g., from a performance [40] 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. Furthermore, as highlighted by Olshannikova et al. [29] the few works existing on big data processes and their visualization highlight a considerable shortcoming in tools and technologies to visualize and interact with data-intensive models at runtime [29].

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 [38], or the validity of the messaging flow in MapReduce [41].

Conclusion

This paper proposes an approach allowing designers and developers to perform analysis of big-data applications by means of code analysis and formal verification techniques. OSTIA provides support to both in the following sense: it helps designers and developers by recovering the architectural topology on-the-fly from the application code and by assisting them in: (a) reasoning on the topological structure and how to refine it; (b) exporting 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 focuses on checking for the compliance to essential consistency rules specific to targeted big data frameworks. (c) Finally, OSTIA allows designers to check whether the recovered topologies contain occurrences of key anti-patterns. By running a case-study with partner organizations, we observed that OSTIA assists designers and developers in establishing and continuously improving the quality of topologies behind their big data applications.

OSTIA can be easily extended to provide more refined tools for the analysis of data-intensive applications as it is general in the approach and modular with respect to the definition of (i) the anti-patterns to be considered and (ii) the formal analysis approaches and the application modeling to be adopted. For this reason, in addition to the practical evidence observed, we believe that OSTIA can be considered as a reference point in the development of data-intensive applications. This motivates us to further elaborate the anti-patterns, exploiting graphs analysis techniques inherited from social-networks analysis. Also, we plan to expand OSTIA to support technologies beyond the most common application framework for streaming and, finally, to further evaluate OSTIA using empirical evaluation.

Abbreviations

OSTIA: ordinary static topology inference analysis; DIA: data intensive application; CIO: Chief-Information Officer; DAG: directed acyclic graph; WICSA: working IEEE/IFIP conference on software architecture; LTL: linear temporal logic; CLTLoc: constraint LTL over clocks; UML: unified modeling language; DICE: developing data-intensive cloud applications with iterative quality enhancement; API: application program interface; ML: machine learning; JSON: javascript object notation; CSV: comma separated variable; XMI: XML metadata interchange (XML—extensible markup language); CLTL: constraint LTL; TA: timed automata; SMT: satisfiability modulo theory; EMF: eclipse modeling framework.

Authors' contributions

All the authors equally contributed to all the sections of the paper. All authors read and approved the final manuscript.

Author details

[1] Politecnico di Milano, Milan, Italy. [2] TU/e - JADS, Eindhoven, The Netherlands. [3] University of South Carolina, Columbia, USA.

Acknowledgements

The authors kindly acknowledge all the people supporting the ideas the allowed the creation of OSTIA.

Competing interests

The authors declare that they have no competing interests

Availability of data and materials

The datasets generated and/or analysed during the current study are available in the [GitHub] repository, (https://github.com/maelstromdat/OSTIA).

Funding

The work is supported by the European Commission Grant No. 0421 (Interreg ICT), Werkinzicht and the European Commission Grant No. 787061 (H2020), ANITA.

Publisher's Note

Received: 29 January 2019 Accepted: 26 April 2019
Published online: 18 May 2019

References

1. http://www.gartner.com/newsroom/id/2637615. Accessed 16 Dec 2013.
2. http://spark.apache.org/. Accessed 1 Dec 2018.
3. https://hadoop.apache.org/. Accessed 1 Dec 2018.
4. https://github.com/maelstromdat/OSTIA. Accessed 1 Dec 2018.
5. http://www.dice-h2020.eu/. Accessed 1 Dec 2018.
6. https://github.com/socialsensor. Accessed 1 Dec 2018.
7. https://github.com/DigitalPebble/storm-crawler. Accessed 1 Dec 2018.
8. https://github.com/sensorstorm/StormCV. Accessed 1 Dec 2018.
9. https://github.com/DigitalPebble. Accessed 1 Dec 2018.
10. Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables devops: an experience report on migration to a cloud-native architecture. 2016.
11. Bersani MM, Distefano S, Ferrucci L, Mazzara M. A timed semantics of workflows. In: ICSOFT (Selected Papers), communications in computer and information Science, vol. 555. Berlin: Springer; 2014. p. 365–83.
12. Bersani MM, Marconi F, Tamburri DA, Jamshidi P, Nodari A. Continuous architecting of stream-based systems. In: Muccini H, Harper EK, editors. Proceedings of the 25th IFIP/IEEE working conference on software architectures. Washington, DC: IEEE Computer Society; 2016. p. 131–42.
13. Bersani MM, Rossi M, San Pietro P. A tool for deciding the satisfiability of continuous-time metric temporal logic. Acta Informatica. 2015:1–36. https://doi.org/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. 2015. arXiv preprint arXiv:1508.04752.
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. New York: ACM; 2014. p. 638–41. https://doi.org/10.1145/2591062.2591088
16. Chandrasekaran K, Santurkar S, Arora A. Stormgen - a domain specific language to create ad-hoc storm topologies. In: FedCSIS. 2014. p. 1621–8.
17. Clements P, Kazman R, Klein M. Evaluating software architectures: methods and case studies. Boston: Addison-Wesley; 2001.
18. Demri S, D'Souza D. An automata-theoretic approach to constraint LTL. Inf Comput. 2007;205(3):380–415.
19. Di Nitto E, Jamshidi P, Guerriero M, Spais I, Tamburri DA. A software architecture framework for quality-aware devops. In: Proceedings of the 2nd international workshop on quality-aware DevOps, QUDOS@ISSTA 2016, Saarbrücken, Germany, July 21, 2016. 2016. p. 12–7. https://doi.org/10.1145/2945408.2945411.
20. Emani CK, Cullot N, Nicolle C. Understandable big data: a survey. Comput Sci Rev. 2015;17:70–81.
21. Evans R. Apache storm, a hands on tutorial. In: IC2E. New York: IEEE; 2015. p. 2.
22. Frankel D. Model driven architecture: applying MDA to enterprise computing. New York: Wiley; 2002.
23. Furia CA, Mandrioli D, Morzenti A, Rossi M. Modeling time in computing: a taxonomy and a comparative survey. ACM Comput Surv. 2010;42(2):6:1–59.
24. Hirzel M, Andrade H, Gedik B, Jacques-Silva G, Khandekar R, Kumar V, Mendell MP, Nasgaard H, Schneider S, Soulé R, Wu KL. Ibm streams processing language: analyzing big data in motion. IBM J Res Dev. 2013;57(3/4):7.
25. Kalantari A, Kamsin A, Kamaruddin H, Ale Ebrahim N, Gani A, Ebrahimi A, Shamshirband S. A bibliometric approach to tracking big data research trends. J Big Data. 2017;4(1):30. https://doi.org/10.1186/s40537-017-0088-1.
26. Krippendorff K. Content analysis: an introduction to its methodology. 2nd ed. Thousand Oaks: Sage Publications; 2004.
27. Marconi F, Bersani MM, Erascu M, Rossi M. Towards the formal verification of DIA through MTL models. In: Lecture notes in computer science.
28. Morgan DL. Focus groups as qualitative research. Thousand Oaks: Sage Publications; 1997.
29. Olshannikova E, Ometov A, Koucheryavy Y, Olsson T. Visualizing big data with augmented and virtual reality: challenges and research agenda. J Big Data. 2015;2(1):22. https://doi.org/10.1186/s40537-015-0031-2.
30. Peng S, Gu J, Wang XS, Rao W, Yang M, Cao Y. Cost-based optimization of logical partitions for a query workload in a hadoop data warehouse. In: Chen L, Jia Y, Sellis TK, Liu G, editors. APWeb, Lecture notes in computer science, vol. 8709. Berlin: Springer; 2014. p. 559–67.
31. Pnueli A. The temporal logic of programs. In: Proceedings of the 18th annual symposium on foundations of computer science, SFCS '77. Washington, DC: IEEE Computer Society; 1977. p. 46–57. https://doi.org/10.1109/SFCS.1977.32
32. Pradella M, Morzenti A, Pietro PS. Bounded satisfiability checking of metric temporal logic specifications. ACM Trans Softw Eng Methodol. 2013;22(3):201–2054. https://doi.org/10.1145/2491509.2491514.
33. Quartulli M, Lozano J, Olaizola IG. Beyond the lambda architecture: effective scheduling for large scale eo information mining and interactive thematic mapping. In: IGARSS. 2015. p. 1492–5.
34. Rajeev A, Dill DL. A theory of timed automata. Theor Comput Sci. 1994;126:183–235.
35. Ratner B. Statistical and machine-learning data mining: techniques for better predictive modeling and analysis of big data. Boca Raton: CRC Press Inc; 2012.
36. Snášel V, Nowaková J, Xhafa F, Barolli L. Geometrical and topological approaches to big data. Futur Gener Comput Syst. 2017;67:286–96. https://doi.org/10.1016/j.future.2016.06.005.
37. Tamura Y, Yamada S. Reliability analysis based on a jump diffusion model with two wiener processes for cloud computing with big data. Entropy. 2015;17(7):4533–46.

38. Tommaso Di Noia MM, Sciascio ED. A computational model for mapreduce job flow. 2014.
39. Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J et al. Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. New York: ACM; 2014. p. 147–56.
40. Wang D, Liu J. Optimizing big data processing performance in the public cloud: opportunities and approaches. IEEE Netw. 2015;29(5):31–5.
41. Yang F, Su W, Zhu H, Li Q. Formalizing mapreduce with csp. In: Proceedings of ECBS. Washington, DC: IEEE Computer Society; 2010. p. 358–67. https://doi.org/10.1109/ECBS.2010.50.