Continuous Architecting of Stream-Based Systems

Marcello Bersani, Francesco Marconi and Damian A. Tamburri
Politecnico di Milano
Milan, Italy
Email: [$marcellomaria.bersani, francesco.marconi, damianandrew.tamburri$]@polimi.it

Pooyan Jamshidi, Andrea Nodari
Imperial College London
London, UK
Email: [$p.jamshidi, a.nodari15$]@imperial.ac.uk

***Abstract*—Big data architectures have been gaining momentum in recent years. For example, Twitter uses stream processing frameworks like Storm to analyse and learn trending topics from billions of tweets per minute. However, maintaining the quality of developed such architectures often requires iterative experiments on expensive multi-node clusters in order to collect performance data to continuously refactor the underlying architecture. As an aid to designers and developers evaluating their applications, we developed OSTIA (On-the-fly Static Topology Inference Analysis) that allows reverse-engineering of topologies for the purpose of: (a) exploiting existing verification techniques on elicited models; (b) visualising and refactoring elicited models while maintaining constraints that would only be evaluated at deployment and run-time; (c) tackling the occurrence of common anti-patterns across considered topologies. We illustrate the uses and benefits of OSTIA on three industrial-scale case studies.**

I. Introduction

Big data applications process large amounts of data for the purpose of gaining key business intelligence through complex analytics such as machine-learning [1], [2]. These applications are receiving increased attention in the last years given their ability to yield competitive advantage by direct investigation of user needs and trends hidden in the enormous quantities of data produced daily by the average Internet user. According to Gartner [1] business intelligence and analytics applications will remain a top focus for CIOs until at least 2017-2018. However, the cost of ownership of the systems that process big data analytics are high due to high infrastructure costs, steep learning curves for the different frameworks involved in designing and developing the applications, such as Apache Storm[2], Apache Spark[3] or Apache Hadoop[4] and complexities in large-scale architectures.

In our own experience with designing and developing for big data, we observed that a key complexity lies in quickly and continuously evaluating the effectiveness of big-data architectures. Effectiveness, in big data terms, means being able to design, deploy, operate, refactor and then (re-)deploy architectures continuously and consistently with runtime restrictions of imposed by the development frameworks. Storm, for example, requires the processing elements to represent a Directed-Acyclic-Graph (DAG). In toy topologies, such constraints can be effectively checked manually, however, when the number of nodes in such architectures increases to real-life industrial scale architecture, it is enormously difficult to verify whether the structural DAG constraints. We argue that this effectiveness can be maintained starting from design time, by enacting a continuous architecting of big data architectures consistently with a DevOps organisational structure [3], [4]. Such structure eases the (re-)deployability of big data architectures and saves the effort to run trial-and-error experiments on expensive infrastructure.

To sustain this argument, we developed OSTIA, that stands for: "On-the-fly Static Topology Inference Analysis". OSTIA allows designers and developers to infer the application architecture through on-the-fly reverse-engineering and architecture recovery [5]. During this inference step, OSTIA analyses the architecture to verify whether it is consistent with restrictions and constraints of the underlying development frameworks (e.g., DAG constraints). In an effort to tackle said complexities in a DevOps fashion, OSTIA is engineered to act as a mechanism that closes the feedback loop between operational data architectures (Ops phase) and their refactoring phase (Dev phase).

Currently, OSTIA focuses on Apache Storm, i.e., one of the most famous and established real-time stream processing engine [6], [7]. The core element of Storm, is called *topology*, which represents the architecture of the processing components of the application (from now we use topology and architecture interchangeably).

OSTIA hardcodes intimate knowledge on the streaming development framework (Storm, in our case) and its dependence structure in the form of a meta-model [8]. This knowledge is necessary to make sure that elicited topologies are correct, so that models may be used in at least four scenarios: (a) realising an exportable visual representation of the developed topologies; (b) verifying structural constraints on topologies that would only become evident during infrastructure setup or runtime operation; (c) verifying the topologies against anti-patterns [9] that may lower performance and limit deployability/executability; (d) finally, use topologies for further analysis, e.g., through model verification [10].

This paper outlines OSTIA, elaborating major usage scenarios, benefits and limitations. Also, we evaluate OSTIA using case-study research and formal validation to conclude that OSTIA does in fact provide valuable insights for continuous

[1] http://www.gartner.com/newsroom/id/2637615
[2] http://storm.apache.org/
[3] http://spark.apache.org/
[4] https://hadoop.apache.org/

architecting.

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

II. Research Design

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

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

This research question emerged as part of our work within the DICE EU H2020 project[5] where we evaluated our case-study owners' scenarios and found that their continuous architecting needs were: (a) focusing on the topological abstractions and surrounding architectural specifications; (b) focusing on bridging the gap between insights from -Ops to (re-)work at the Dev- level; (c) their needs primarily consisted in maintaining framework consistency during architecture reworks. In pursuit of the research question above, the results contained in this paper were initially elaborated within a free-form focus group [11] involving three experienced practitioners and researchers on big data streaming technologies, such as Storm. Following the focus group, through self-ethnography [12] and brainstorming we identified the series of essential consistency checks, algorithmic evaluations as well as anti-patterns that can now be applied through OSTIA while recovering an architectural representation for Storm topologies. We designed OSTIA[6] to support the incremental and iterative refinement of streaming topologies based on the incremental discovery and correction of the above checks and patterns.

Finally, OSTIA's evaluation is threefold. First, by means of an industrial case-study offered by one of the industrial partners in the DICE EU H2020 Project consortium[7]. 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[8] which features the combined action of three complex streaming topologies based on Apache Storm (see Fig. 12 for a sample). In particular, the topology in Fig. 12 extracts data from sources and manipulates said data to divide and arrange contents based on type (e.g., article vs. media), later updating a series of databases (e.g., Redis) with these elaborations. The models that OSTIA elicited from this application were showcased to our industrial partner in a focus group aimed at establishing the value of insights produced as part of OSTIA-based analyses. Our qualitative assessment was based on questionnaires and open discussion.

Second, to further confirm the validity of OSTIA analyses and support, we applied it on two open-source applications featuring Big-Data analytics and built on top of the Storm streaming technology, namely: (a) the DigitalPebble application, i.e., quoting from the homesite[9], "A Text Classification API in Java originally developed by DigitalPebble Ltd. The API is independent from the ML implementations used and can be used as a front end to various ML algorithms"; (b) the StormCV application, i.e., quoting from the homesite[10] "StormCV enables the use of Apache Storm for video processing by adding computer vision (CV) specific operations and data model; the platform enables the development of distributed video processing pipelines which can be deployed on Storm clusters".

Third, finally, we applied well-established verification approaches to integrate the value and benefits behind using OSTIA. We engineered OSTIA to support exporting of elicited topologies for their further analysis using the Zot[11] LTL model-checker using an approach outlined in our previous work [10].

III. Research Solution

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

A. Storm architecture

Storm is a technology developed at Twitter [7] in order to face the problem of processing of streaming of data. It is defined as a distributed processing framework which is able to analyse streams of data. The core element in the system is called *topology*. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts. The former type includes nodes that process the data entering the topology, for instance querying APIs or retrieve information from a message broker, such as Apache Kafka. The latter executes operations on data, such as filtering or serialising.

B. OSTIA design

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

[5]http://www.dice-h2020.eu/

[6]https://github.com/maelstromdat/OSTIA

[7]http://www.dice-h2020.eu/

[8]https://github.com/socialsensor

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

[10]https://github.com/sensorstorm/StormCV

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

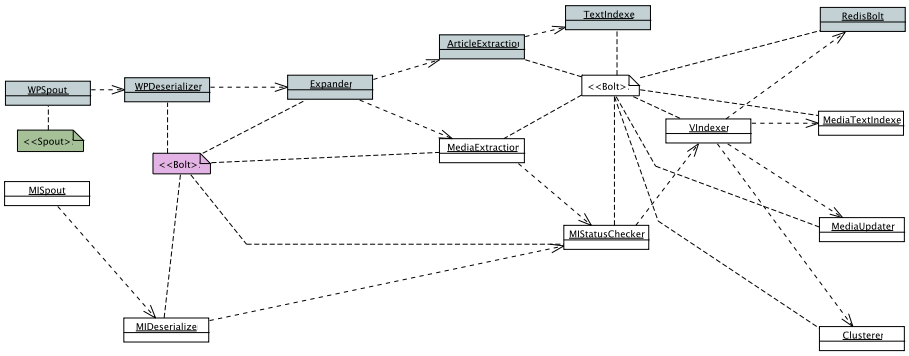

Fig. 1. A sample Storm topology in the SocialSensor App.

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

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

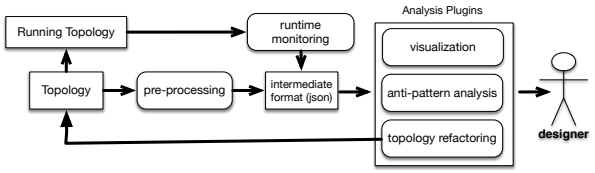

Fig. 2. OSTIA extensible architecture.

Such information will be continuously added to the intermediate repository via runtime monitoring of the topology on real deployment cluster. These provide appropriate and rich information for refactoring the initial architecture and enabling performance driven DevOps [13].

Finally, OSTIA allows users to export the topology in different formats (e.g. JSON) to analyse the topology with other tools.

C. Deep Within OSTIA: The Storm Framework Meta-Model

OSTIA was designed to retrieve and analyse Storm topologies on-the-fly, allowing their refactoring in a way which is consistent with framework restrictions, rules and regulations part of the Storm framework. In doing so, OSTIA uses a meta-model for the Storm framework. This meta-model acts as an image of all said restrictions and rules that OSTIA needs to maintain for elicited (or refactored) topologies. Essentially OSTIA implements in the meta-model the operational picture of the Storm Framework - this is critical to checking that the restrictions and constraints coded within Storm are reflected in elicited models as well as maintained during architecting and refactoring of those models. In addition, OSTIA applies all the necessary anti-pattern checks in combination with checking that said framework restrictions are maintained - this is critical to support continuous architecting in a manner which is consistent with Storm restrictions that would only become apparent during run-time and operations. The meta-model in question is depicted in Fig. 3. The figure shows an overview of the meta-model for Storm[12] where, for example, the grouping restrictions that Storm envisions are captured in an enumeration of constraints (see the <<Grouping>> element or the <<ReplicationFactor>> concrete parameter). Key elements of the meta-model are the following:

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

IV. Topology Design Anti-Patterns

This section and section V elaborate on the Anti-patterns and algorithmic analysis supported in OSTIA. All figures in these sections use a simple graph-like notation where nodes may be any topological element (e.g., Spouts or Bolts in Apache Storm terms) while edges are to be interpreted as directed data-flow connections.

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

[12]The details of this meta-model and the restrictions captured therein is beyond the scope of this paper. More details are available in our project homepage, in section D2.1: http://dice-h2020.eu/deliverables/D2.1.

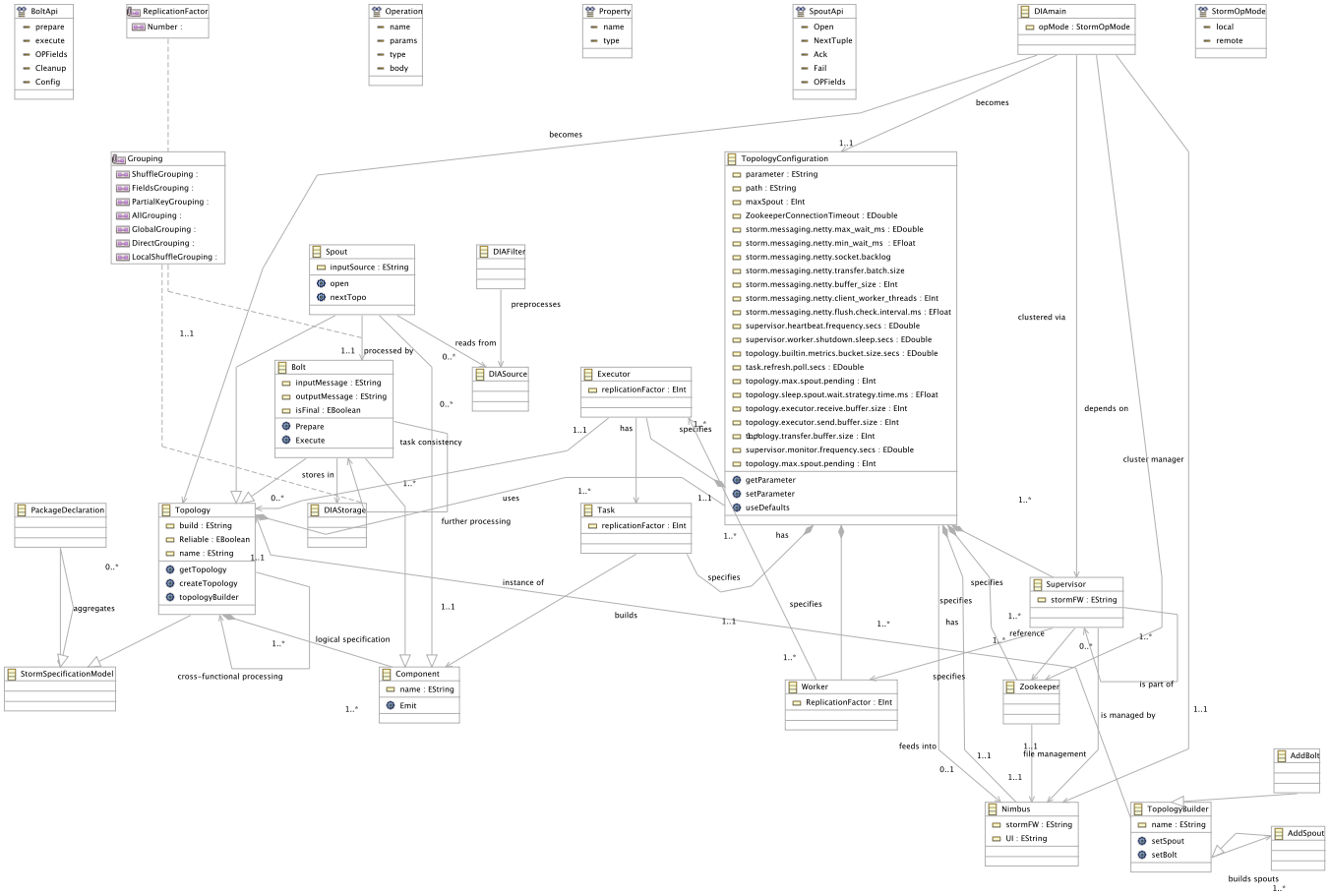

Fig. 3. Deep Within OSTIA, a Storm Meta-Model.

A. Multi-Anchoring

In order to guarantee fault-tolerant stream processing, tuples processed by bolts needs 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.

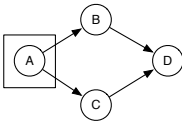

Fig. 4. Multi-anchoring.

B. Cycle-in Topology

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

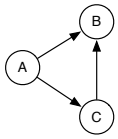

Fig. 5. Cycle-in Topology.

C. Persistent Data

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 (see Sec. VII-B).

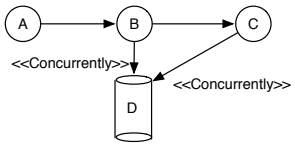

Fig. 6. Concurrency management in case of Persistent Data circumstances.

D. Computation funnel

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.

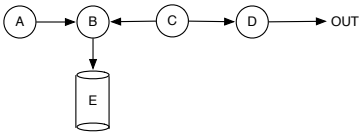

Fig. 7. computation funnel.

V. Algorithmic Analysis on Stream Topologies

A. fan-in/fan-out

For each element of the topology, fan-in is the number of incoming streams. Conversely, fan-out is the number outgoing streams. In the case of bolts, both in and out streams are internal to the topology. For Spouts, incoming streams are the data sources of the topology (e.g., message brokers, APIs, etc).

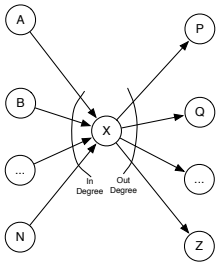

Fig. 8. Fan-in fan-out in Stream topologies.

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

B. topology cascading

By topology cascading, we mean connecting two different Storm topologies via a messaging framework (e.g., Apache Kafka). This circumstance, which is actually part of our evaluation and case-studies, may raise the complexity of the overarching topology to unacceptable levels and may require additional attention. OSTIA support for this feature is still limited, more details on this and similar limitations are discussed in Section VII-B.

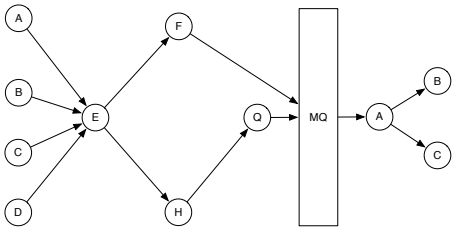

Fig. 9. cascading.

This algorithmic manipulation allows to combine multiple cascading topologies.

C. Topology clustering

Identifying the coupled processing elements and put the in a cluster in a way that elements in a cluster have high cohesion and less coupled with the elements in other clusters. Simple clustering or Social-Network Analysis mechanisms can be used to infer clusters. These clusters may require additional attention since they could turn out to become bottlenecks. Reasoning more deeply on clusters and their resolution may lead to establishing the Storm scheduling policy best-fitting with the application at hand.

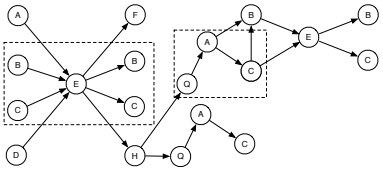

Fig. 10. clustering.

D. Linearising a topology

Sorting the processing elements in a topology in a way that topology looks more linear, visually. This step ensures that visual investigation and evaluation of the structural complexity of the topology is possible by direct observation. It is sometimes essential to provide such a visualisation to evaluate how to refactor the topology based on emerging needs.

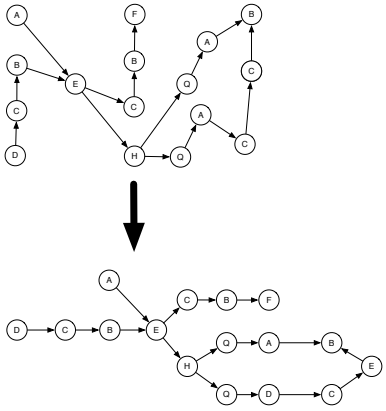

Fig. 11. linearizing.

VI. Evaluation

This section elaborates on our evaluation campaign. As previously stated, we evaluated OSTIA through qualitative evaluation and case-study research featuring an open-/closed-source industrial case-study (see Section VI-A) and two open-source case-studies (see Section VI-B) on which we also applied complex formal verification (see Section VI-C).

A. Industrial Case-Study

As previously introduced in Section II, we operated an evaluation of OSTIA running the application on a closed-source cascade of several topologies part of the SocialSensor App. Our industrial partner is having performance and availability outages connected to currently unknown circumstances. Therefore, the objective of our evaluation for OSTIA was twofold: (a) allow our industrial partner to enact continuous architecting of their application with the goal of discovering any patterns or hotspots that may be requiring further architectural reasoning; (b) understand wether OSTIA provided valuable feedback to endure the continuous architecting exercise.

OSTIA standard output[13] for the smallest of the three SocialSensor topologies, namely the "focused-crawler" topology, is outlined in Fig. 12.

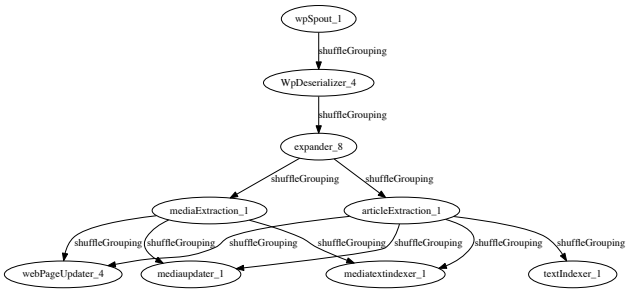

Fig. 12. SocialSensor App, OSTIA sample output.

[13]output of OSTIA analyses is not evidenced for the sake of space.

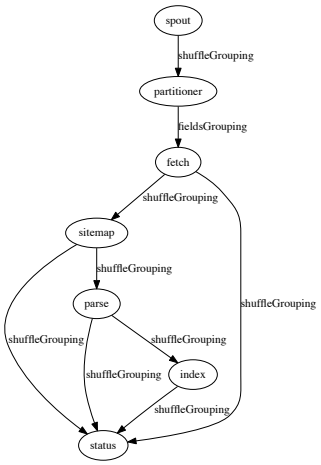

Fig. 13. DigitalPebble application topology, an overview.

OSTIA findings are as follows:

- A double instance of the "Multi-Anchoring" pattern is found around bolt "expander";
- An instance of the "Persistent Data" pattern is found around bolt "expander" as well;
- An instance of the "Computational Funnel" pattern is found around bolt "expander";

as a consequence of these outputs our industrial partner observed that the "expander" bolt clearly needed additional architectural reasoning. Also, the partner welcomed the idea of using OSTIA as a mechanism to enact continuous architecting of the topology in question as part of the needed architectural reasoning.

Besides this pattern-based evaluation and assessment, OSTIA algorithmic analyses assisted our client in understanding that the topological structure and cascade 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[14] framework for batch processing.

B. Evaluation on Open-Source Software

In order to confirm the usefulness and capacity of OSTIA to enact a continuous architecting cycle loop, we applied it in understanding (first) and attempting improvements of two open-source applications, namely, the previously introduced DigitalPebble[15] and StormCV[16] applications. Figures 13 and 14 outline standard OSTIA output for the two applications.

The OSTIA output aided as follows. First, the output summarised in Fig. 13 allowed us to immediately grasp

[14]http://hadoop.apache.org/

[15]https://github.com/DigitalPebble

[16]https://github.com/sensorstorm/StormCV

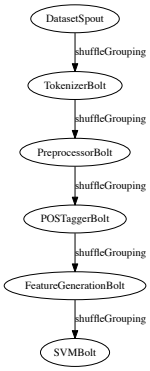

Fig. 14. StormCV application topology, an overview.

the functional behavior of the DigitalPebble and StormCV topologies allowing us to interpret correctly their operations before reading long documentation or inspecting the code. Also, thanks to OSTIA, we found that DigitalPebble shows several instances of OSTIA anti-patterns, e.g., the "Cycle-in" and "multi-anchoring" patterns in several places around the topology. In addition, we observed that the StormCV architecture seems fairly simple. However, checking the code for the StormCV bolts and spouts we found that said code was fairly dense and therefore the resulting topology may need fine-tuning thorugh additional "divide-et-impera" architectural refinements.

C. OSTIA-Based Formal Verification

This section describes the formal modelling and verification employed in OSTIA which both rely on *satisfiability checking* [14], an alternative approach to model-checking. Instead of an operational model (like automata or transition systems), as in model-checking, the system (i.e., a topology in this context) is specified by a formula defining their executions over time and properties are verified by proving that the system logically entails them.

The logic we use is Constraint LTL over clocks (CLTLoc) [15] which is a semantic restriction of Constraint LTL (CLTL) [16] allowing atomic formulae over $(\mathbb{R}, \{<, =\})$ where the arithmetical variables behave like clocks of Timed Automata (TA) [17]. A clock x measures the time elapsed since the last time when $x = 0$ held, i.e., since the last "reset" of x. Clocks are interpreted over Reals and their value can be tested with respect to a positive integer value. Let X be a finite set of clock variables x over $\mathbb{R}$ and AP be a finite set of atomic propositions p. CLTLoc formulae are defined as follows:

$$\phi := p \mid x \sim c \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}(\phi) \mid \mathbf{Y}(\phi) \mid \phi \mathbf{U} \phi \mid \phi \mathbf{S} \phi$$

where $c \in \mathbb{N}$ and $\sim \in \{<, =\}$, $\bullet$, $\circ$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since". A *model* is a pair (π, σ), where σ is a mapping associating every variable x and position in $\mathbb{N}$ with value $\sigma(i, x)$ and π is a mapping associating each position in $\mathbb{N}$ with subset of AP. The semantics of CLTLoc is defined as for LTL except for formulae $x \sim c$. At position $i \in \mathbb{N}$, $(\pi, \sigma), i \models x \sim c$ **iff** $\sigma(i, x) \sim c$. A formula is *satisfiable* if it has a model.

The standard technique to prove the satisfiability of CLTL and CLTLoc formulae is based on of Büchi automata [16], [15] but, for practical implementation, Bounded Satisfiability Checking (BSC) [14] avoids the onerous construction of automata. By unrolling the semantics of a formula for a finite number $k > 0$ of steps, the outcome of a BSC problem is either an infinite ultimately periodic model or unsat. [15] shows that BSC for CLTLoc is complete and that is reducible to a decidable Satisfiability Modulo Theory (SMT) problem. A CLTLoc formula can be translated into the decidable theory of quantifier-free formulae with equality and uninterpreted functions combined with the theory of Reals over $(\mathbb{R}, <)$.

CLTLoc allows the specification of temporal constraints using clock variables ranging over $\mathbb{R}$, whose value is not abstracted. Clock variables represent, in the logical language and with the same precision, physical (dense) clocks. They appear in formulae of the form $x \sim c$ to express a bound c on the delay measured by clock x. Clocks are associated with specific events to measure time elapsing over the execution. 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.

D. Storm Formal Model

To perform our verification tasks we defined a formal model expressed in CLTL over clocks with discrete variables. The resulting model is a non-deterministic infinite state system.

1) OSTIA Models: a Formal Interpretation: We started by understanding and capturing the behaviors of both spouts and bolts. After choosing the level of abstraction of our model we simplified those behaviors accordingly, in order to formalize them as finite state machines. The purpose of this first activity was to define the possible operations and the allowed orderings of such operations. We then extended the model by taking into account the message buffers (or queues) and the quantity of tuples that are exchanged through the topology. In addition to the correct ordering of the operations, we decided to introduce more specific temporal constraints into the model, in order to limit the time spent by the system in each state (or processing phase) and to elaborate the concept of *rate*, intended as "number of times an event is occurring every time unit".

2) Assumptions and level of abstraction: We made several assumptions and abstractions while building the model:

- we abstracted away some deployment-related details, such as the number of worker nodes and characteristics of the underlying cluster;
- each bolt/spout has a single output stream;

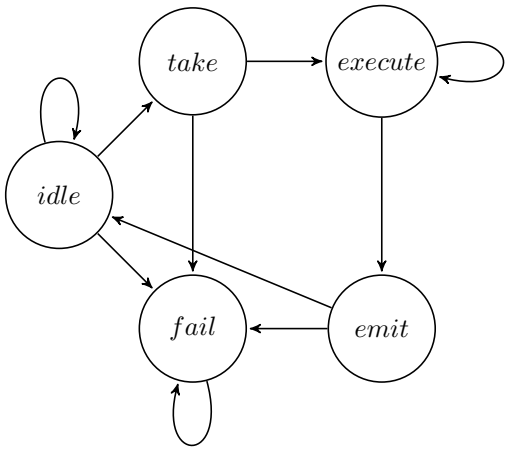

Fig. 15. Finite state automaton describing bolt states.

- we simplified the message buffer system, assuming that there is a single queuing layer: every bolt has a single incoming queue and no sending queue, while the worker queues are not represented;
- every operation is performed within minimum and maximum thresholds of time;
- we do not take into account the content of the messages: all the tuples have same fixed size and we represent only quantity of tuples moving through the system;

3) Model Formalization: A Storm Topology is a directed graph $\mathbf{G} = \{\mathbf{N}, \mathbf{E}\}$ such that the set of nodes $\mathbf{N} = \mathbf{S} \bigcup \mathbf{B}$ includes in the sets of spouts (**S**) and bolts (**B**), while the set of edges $\mathbf{E} = \{Sub_{i,j} | i \in \mathbf{B}, j \in \{\mathbf{S} \bigcup \mathbf{B}\}\}$ defines how the nodes are connected each other via the subscription relationship ($Sub_{i,j}$). $Sub_{i,j}$ indicates that "bolt i subscribes to the streams emitted by the spout/bolt j". Follows that 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 by the node. The queues have infinite size and the level of occupation of each j^{th} queue is described by the variable q_j.[17]
Each spout can either *emit* tuples into the topology or stay idle. Each bolt can be in an idle state, in a failure state or in a 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.

To give an idea about how the model is formalized, we provide 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".*

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

The number of tuples emitted by a bolt depends on the number of incoming tuples. The ratio $\frac{\#output\_tuples}{\#input\_tuples}$ is used to express 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".*

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

These functional constraints are fixed for all the nodes and not configurable. What is configurable and can be tuned changing the parameters of the model is everything concerning 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.
Currently the verification tool accepts as configuration format a JSON file containing all the needed 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.

VII. Discussion

This section discusses our findings and limitations around OSTIA.

A. Findings and Continuous Architecting Insights

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

First, we found (and observed in industrial practice) that it is often more useful to develop a quick-and-dirty but "runnable" architecture topology than improving the topology at design time for a tentatively perfect execution.This is mostly the case with big-data applications that are developed stemming from previously existing topologies or applications. OSTIA hardcodes this way of thinking by supporting reverse-engineering and recovery of deployed topologies for their incremental improvement. Although we did not carry out extensive qualitative or quantitative evaluation of OSTIA in

[17]Spouts have no queues, by definition.

this regard, we are planning additional industrial experiments for future work with the goal of increasing OSTIA usability and practical quality.

Second, big-data applications design is an extremely young and emerging field for which not many software design patterns have been discovered yet. The (anti-)patterns and approaches currently hardcoded into OSTIA are inherited from related fields, e.g., pattern- and cluster-based graph analysis. Nevertheless, OSTIA may also be used to investigate the existence of recurrent and effective design solutions (i.e., design patterns) for the benefit of big-data application design. We are improving OSTIA in this regard by experimenting on two fronts: (a) re-design and extend the facilities with which OSTIA supports anti-pattern detection; (b) run OSTIA on multiple big-data applications stemming from multiple technologies beyond Storm (e.g., Spark, Hadoop Map Reduce, etc.) with the purpose of finding recurrent patterns. A similar approach may feature OSTIA as part of architecture trade-off analysis campaigns [18].

Third, finally, 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 [19]. OSTIA may aid in this investigation by allowing the interactive and incremental improvement of multiple (combinations of) topologies together.

B. Approach Limitations and Threats to Validity

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

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

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

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

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

VIII. Related Work

The work behind OSTIA stems from the EU H2020 Project called DICE[18] 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 [20] provides an implementation language specific to programming streams management (e.g., Storm jobs) and related reactive systems based on the Big-Data paradigm.

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 [21] or reliability point of view [22]. Said works use complex math-based approaches to evaluating a number of Big data architectures, their structure and general configuration. However, although novel, these approaches do not suggest any significant design improvement method or pattern to make the improvements *deployable*. With OSTIA, we make available a tool that automatically elicits a Storm topology and, while doing so, analyses said topology to evaluate it against a number of consistency checks that make the topology consistent with the framework it was developed for (Storm, in our case). To the best of our knowledge, no such tool exists to date.

Third, from a modelling perspective, approaches such as StormGen [23] 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 and developers can work hand in hand while 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 Storm topologies.

Fourth, from a verification perspective, to the best of our knowledge, this represents the first attempt to build a formal

[18]http://www.dice-h2020.eu/

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 [24], 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 [25], or the validity of the messaging flow in MapReduce [26][19].

IX. Conclusion

Applications that make heavy use of Big data application frameworks require intensive reasoning and continuous architecting of design aspects typically around the topology of the basic operations to be applied on the data. We set out to assist the continuous architecting of Big-Data streaming designs by OSTIA, a toolkit to assist designers and developers in this continuous architecting campaign. 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 we observed in our own experience and previous work. 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 in multiple ways. We confirmed this result running OSTIA on several open-source applications featuring streaming technologies.

In the future we plan to further our understanding of the anti-patterns that may emerge across big data topologies, e.g., as discussed, by learning said anti-patterns by using graphs analysis techniques inherited from social-networks analysis. Also, we plan to expand OSTIA to support further technologies beyond the most common application framework for streaming, i.e., Storm. Finally, we plan to further evaluate OSTIA using more ad-hoc empirical evaluation in industry.

References

[1] C. K. Emani, N. Cullot, and C. Nicolle, "Understandable big data: A survey." *Computer Science Review*, vol. 17, pp. 70–81, 2015.

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

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

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

[4] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective.*, ser. SEI series in software engineering. Addison-Wesley, 2015.

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

[6] R. Evans, "Apache storm, a hands on tutorial." in *IC2E*. IEEE, 2015, p. 2.

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

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

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

[10] M. M. Bersani, S. Distefano, L. Ferrucci, and M. Mazzara, "A timed semantics of workflows." in *ICSOFT (Selected Papers)*, ser. Communications in Computer and Information Science, A. Holzinger, J. Cardoso, J. Cordeiro, T. Libourel, L. A. Maciaszek, and M. van Sinderen, Eds., vol. 555. Springer, 2014, pp. 365–383.

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

[12] M. Hammersley and P. Atkinson, *Ethnography*. London: Routledge, 2003.

[13] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski *et al.*, "Performance-oriented devops: A research agenda," *arXiv preprint arXiv:1508.04752*, 2015.

[14] M. Pradella, A. Morzenti, and P. S. Pietro, "Bounded satisfiability checking of metric temporal logic specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 20:1–20:54, Jul. 2013.

[15] M. M. Bersani, M. Rossi, and P. San Pietro, "A tool for deciding the satisfiability of continuous-time metric temporal logic," *Acta Informatica*, pp. 1–36, 2015.

[16] S. Demri and D. D'Souza, "An automata-theoretic approach to constraint LTL," *Information and Computation*, vol. 205, no. 3, pp. 380–415, 2007.

[17] Rajeev Alur and David L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, p. 183?235, 1994.

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

[19] M. Quartulli, J. Lozano, and I. G. Olaizola, "Beyond the lambda architecture: Effective scheduling for large scale eo information mining and interactive thematic mapping." in *IGARSS*. IEEE, 2015, pp. 1492–1495.

[20] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. P. Mendell, H. Nasgaard, S. Schneider, R. Soul, and K.-L. Wu, "Ibm streams processing language: Analyzing big data in motion." *IBM Journal of Research and Development*, vol. 57, no. 3/4, p. 7, 2013.

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

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

[23] K. Chandrasekaran, S. Santurkar, and A. Arora, "Stormgen - a domain specific language to create ad-hoc storm topologies." in *FedCSIS*, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., 2014, pp. 1621–1628.

[24] M. Camilli, "Formal verification problems in a big data world: Towards a mighty synergy," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 638–641.

[25] T. Di Noia, M. Mongiello, and E. Di Sciascio, "A computational model for mapreduce job flow."

[26] F. Yang, W. Su, H. Zhu, and Q. Li, "Formalizing mapreduce with csp," in *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*. IEEE, 2010, pp. 358–367.