







# An Inspection and Logging System for Complex Event Processing in Bosch's Industry 4.0 Movement

Carina Andrade<sup>(✉)</sup> , Maria Cardoso , Carlos Costa ,  
and Maribel Yasmina Santos 

ALGORITMI Research Centre, University of Minho, Guimarães, Portugal  
{carina.andrade, carlos.costa, maribel}@dsi.uminho.pt,  
a78439@alunos.uminho.pt

**Abstract.** Currently, it is possible to have machines producing relevant data to be processed in real-time, facilitating the organizational decision-making. In recent works, we proposed a system that integrates Complex Event Processing (CEP) in the Big Data era, trying to make Industry 4.0 systems more pro-active. Due to its complexity when running in industrial contexts, appropriate monitoring mechanisms need to be ensured to prevent the uncontrolled growth of the system. In this context, this work focuses on proposing a system architecture that will enable an innovative monitoring strategy based on graph analysis, namely the *Intelligent Event Broker (IEB)* Mapping and Drill-down System. In this work, it is proposed an inspection and logging strategy for the *IEB* that allows to not only continuously inspect the codebase of the system and fuel an ever-growing Graph Database, but also to strategically store log occurrences to know what is continuously happening. For demonstrating the architecture and design rules, we use a context from Bosch Portugal presenting a flowchart and a graph data model, being the latter a mirror of all the implemented *IEB* components and the relationships between them. This work helps researchers and practitioners in the design and development of CEP systems for Big Data contexts and, especially, the monitoring component of such a complex system.

**Keywords:** Big data · Complex event processing · Monitoring · Graph database

## 1 Introduction

Nowadays, several industries are pursuing the adoption of Big Data and Real-time concepts in their enterprises. This need arose, for example, from the current technological evolution that results on a huge amount of data being produced every day by various types of machines inside the shop floor. Once the data is available, the challenge is how to use it to improve the organizations' performance by acting intelligently and without need to wait for human analysis and approvals. The Complex Event Processing (CEP) concept has existed since the 90s, always linked to the need to process various events in a real-time fashion. Nowadays, CEP is being integrated into Big Data contexts due to the need to

process events resulting from real-time data streams that are more frequently available in the organizations.

The *Intelligent Event Broker (IEB)* is proposed in [1] as an innovative system that integrates the CEP and Big Data concepts using a Rules Engine embedded into Spark<sup>1</sup>. The architecture considers the existence of several types of data sources and a component (*Producers*) dedicated to the standardization of the connection to all of them. The events arriving at the system are serialized into classes representing the business entities (*Broker Beans*) that will be subscribed by the *Event Processor* (Spark *Consumers*). This last component can send event data to be aggregated for further Key Performance Indicators (KPIs) calculation in the *Event Aggregator* (supported by Druid<sup>2</sup>). *Consumers* are also directly related to the *Rules Engine* (implemented in Drools<sup>3</sup>) where all the business requirements are defined as strategical, tactical, or operational *Rules*. These three types of *Rules* are translated at runtime by the *Consumers*. The *Triggers* component represents the connection to all the *Destination Systems*, and they will perform certain actions based on the results of the rules verification (e.g., stop a production machine if the last three products registered a failure). The *Predictors and Recommenders* is the component responsible for the application of previously trained Machine Learning models, which are stored in the *Lake of Machine Learning Models*.

Furthermore, it is considered that this kind of system needs closer and rich monitoring capabilities. In this context, a *Mapping and Drill-down System* was previously included in the *IEB* architecture [1], considering a *Graph Database* and a *Web Visualization Platform* to perform the system monitorization. This *Graph Database* will store the data related to the *IEB* codebase that is continuously and automatically inferred, as well as the relationships between them and the logs from the system components that are continuously running. The collected data will be analyzed in a *Web Visualization Platform* already proposed in [2]. Therefore, this work aims to detail this system by proposing an architecture and a set of design rules that will ensure the adequate data collection to feed the *Mapping and Drill-down System*, ensuring the efficient inspection and logging of the *IEB*.

This paper is structured as follows: Sect. 2 presents the related work identified in the literature; Sect. 3 presents the inspection and logging system architecture and a set of design rules; Sect. 4 presents the demonstration case to highlight how the inspection component of the system was implemented, using the Bosch Portugal context; Sect. 5 presents the conclusions and future work.

## 2 Related Work

Being this a very specific topic inside what is already a very specific research context (i.e., CEP on Big Data contexts), there is a lack of relevant literature and related works. Regarding the existence of systems that integrate the CEP concept in Big Data contexts,

<sup>1</sup> <https://spark.apache.org/>.

<sup>2</sup> <https://druid.apache.org/>.

<sup>3</sup> <https://www.drools.org/>.

few works were found in the literature. The work of [3] proposes an architecture (BiDCEP) for a system that integrates the CEP capabilities in the Big Data world. For that, the authors idealized a mixed Big Data Streaming architecture based on the recognized Lambda and Kappa architectures for Big Data. This proposal is summarized as being the extension of these architectures with components that represent the CEP system. The work of [4] presents a prototype named FERARI that aims to process a large number of event streams in a multi-cloud environment. Their proposal is based on four components, each one with distinct goals: a web-based graphical user interface to define CEP concepts; a component to plan the latency and communication between the instances of the inter-cloud; a component responsible for the events processing and a web-based dashboard with reports regarding the processed events. Another architecture to integrate CEP and Big Data using only open source technologies is discussed in [5], using an electronic coupon distribution centre as a demonstration case and giving focus to the technologies selection (uses Apache Kafka as a message broker, HDFS to store the data and a CEP system based on If-Then-Else rules to process the data).

During the analysis of the related work, when looking for the need of systems to monitor a CEP in Big Data contexts, works only revealed the possible use of CEPs to monitor other systems [6, 7]. In this context, proper logging was considered since logs are widely used to indicate the state of the system at runtime, providing the details of the transactions that occur and containing useful information (e.g., name, date, and time of the occurrence [8]), which helps to understand the behaviour of the system. The work of [9] mentions that a log must be recorded in an orderly and controlled manner so that it is human-readable. Nevertheless, its usefulness depends on how the log is applied, proposing much more than debug information and being of considerable value when analyzing the performance of an application [9].

Regarding the monitorization of this kind of systems, this concern was only identified in the FERARI project, with a dashboard component that provides reports about some metadata of the system (e.g., daily events or for the last 4 h) [10], being the focus on the data flows of the system and apparently leaving aside the drill-down into the insights and behaviour from the individual components of the system.

### 3 The Inspection and Logging System as a *IEB* Mapping and Drill-Down Feeder

Considering the complexity that can arise with the evolution of a system like the *IEB* proposed in [1], especially when running it in industrial contexts, a dedicated system must work in parallel to guarantee the constant and long-term monitoring of the *IEB*'s daily operations, ensuring its sustainable and controlled growth.

Taking this into consideration, the *Mapping and Drill-down System* was included in [1] and considers the implementation of two components: i) a *Graph Database* (considered the most adequate database due to the need to deal with the constant evolution of the *IEB* and its potential growth when running with several subjects simultaneously); and, ii) a *Web Visualization Platform* to enable the drill-down over the data, as discussed in [2]. This work is focused on the design of the system that will allow the fueling of the *Graph Database*, to operate as the data source for the *Web*

**Visualization Platform**, allowing the *IEB* stakeholders and operators to establish relevant relationships and drill-down operations into several occurrences within the system and its components.

### 3.1 System Architecture

As briefly highlighted in [1], the inspection and logging system architecture present here must ensure the following goals: **G1**) the proper analysis and indexing of the *IEB* codebase; **G2**) appropriate runtime logging mechanisms, to guarantee the storage of all the relevant data about the system components that are constantly working; **G3**) the analysis of the system functionality, recording what happened and when; and, **G4**) the analysis of the system performance (e.g., how many events were produced in *Producer X*, or consumed by *Consumer Y*). To ensure that all these defined goals are considered, the system architecture presented in Fig. 1 is divided into two parts:

1. The first part is dedicated to code inspection. For that, the *IEB* codebase (*System Code Repository*) is used as the source for analysis, exploring the folders and files that compose the system and using the *Code Inspector* component to collect data to feed the *Graph Database* (e.g., collect data to create graph labels, graph nodes, graph nodes' properties and relationships between graph nodes). To collect all the relevant data that will allow the definition of the Graph Data Model (GDM), a set of design rules were defined and are presented in Subsect. 3.2. The implementation of these rules will guarantee the creation of the GDM regardless of how the system is implemented and addressing the previously presented goal **G1**);
2. The second part is related to logging mechanisms. Here, the *IEB* components already implemented are considered as data sources to provide useful data when they are running. The *Logger* component should be embedded in the *IEB* components, logging the time, the events flowing through the system, and the system components execution. The time logs are the key point since time will be a property in the relationships between graph nodes, representing when something happened in the system. Secondary storage is proposed for historical and analytical purposes due to the dimension that the *Graph Database* can achieve.
  - a. In the *Historical Storage*, all the raw data history will be available for analytical purposes, when ad hoc queries involving the use of complex interactions and calculations is needed. Here, analytical tools like Hive<sup>4</sup> or Spark can be used to explore the data in the Hadoop Distributed File System (HDFS), and data exploration tools like Zeppelin<sup>5</sup> or Tableau<sup>6</sup> can be used to interact visually with all the raw historical data.
  - b. In *Interactive Storage*, a graph database has the most recent data to drill-down over it and take advantage of the analysis that can be done on graphs. The time frame defined for the data stored in the graph database depends on three factors:

<sup>4</sup> <https://hive.apache.org/>.

<sup>5</sup> <https://zeppelin.apache.org/>.

<sup>6</sup> <https://www.tableau.com/>.

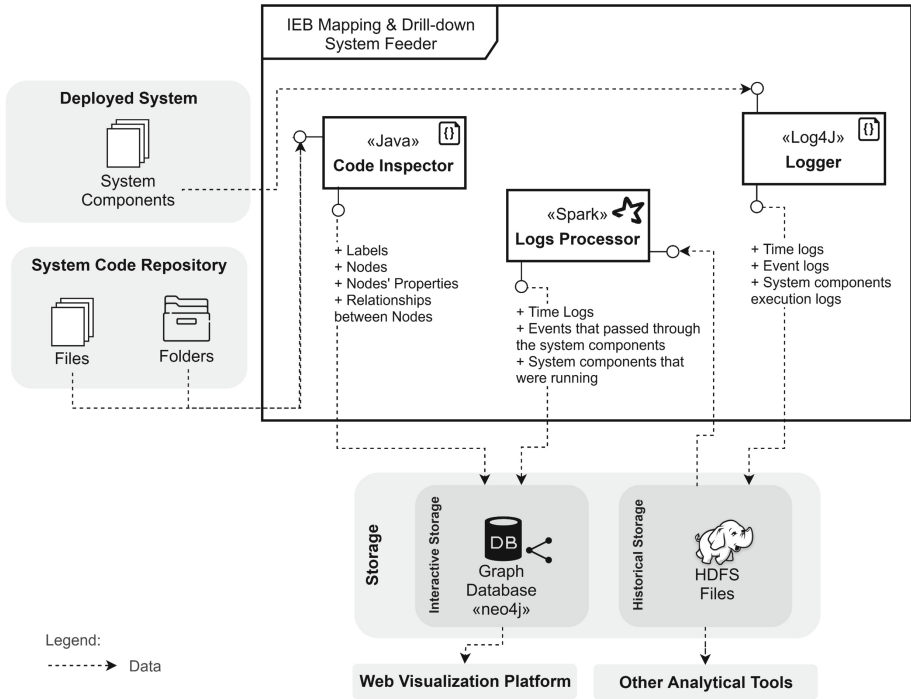


Fig. 1. System architecture

i) the business requirements; ii) the amount of data generated by the system; and, iii) the capacity of the neo4j infrastructure. The *Logs Processor* component will pick up the data from the *Historical Storage* (at predefined periods) to load it into the graph database (*Interactive Storage*). This will guarantee the achievement of the previously presented goals **G2**, **G3**) and **G4**).

The architecture presented in Fig. 1 already proposes technologies that can be used for each component (e.g., Spark for the *Logs Processor* component), considering the technological stack already proposed in [1] for the remaining system. Regarding the *Logger* component, Log4j<sup>7</sup> was proposed considering that is an open-source structure, flexible, written in Java and currently commonly used [11]. However, practitioners are free to choose other technologies for their specific implementation of a similar system to monitor their CEP system in Big Data contexts, as the conceptual proposal still holds true.

### 3.2 Design Rules for the Code Inspector Component

Related to the repository code inspection, several design rules are defined to guarantee that all the relevant *IEB* components are identified and tracked appropriately, as well as

<sup>7</sup> <http://logging.apache.org/log4j/2.x/>.

their relationships. These design rules take into consideration that the result of the code inspection should be a GDM to be implemented in a graph database. Once these design rules are implemented in a piece of software dedicated to inspecting the *IEB*, in any programming language practitioners choose to (ours is in Java due to the codebase of the *IEB*), all the relevant inspection data will be captured to create the graph (namely the inspection part of the graph) for the *Mapping and Drill-down System*. The defined design rules (DR) are presented as follow:

**DR1: The *IEB* components are the 1<sup>st</sup> level labels.** This will guarantee that all the implemented components from the *IEB* (mentioned in Sect. 1) are categorized by their type of component in the architecture, facilitating the future exploration of the graph database (e.g., *Producers*, *Rules* or *Triggers*).

**DR2: The folders' name, where the components were found, are 2<sup>nd</sup> level labels, if it defines a specific subject for the *IEB*.** Considering the variety of subjects that can be implemented in the *IEB*, a clear separation by folders should be made during the implementation. The collection of this information will enable the differentiation between the several topics implemented in the system (e.g., “/producers/alr” or “/consumers/shop-floor-incidents”).

**DR3: The files' name that reflects the implementation of *IEB* components are the names of the graph nodes.** The graph nodes will be the several instantiations of the system components already implemented for the various *IEB* contexts (e.g., *Producers* or *Rules* for a subject).

**DR4: The labels defined in DR1 and DR2 must be associated to the graph nodes identified in DR3.** This will guarantee that all the created graph nodes will have at least one label associated, identifying the system component and (in some cases) the subject related to their implementation.

**DR5: In the *Broker Beans*' files, their variables and data types are their graph nodes' properties.** Considering the importance of this component when representing the data that will flow throughout the system, it is relevant that the *Broker Beans* variables and respective data types are collected.

**DR6: In the *Broker Beans*' files, if the variable data type is another *Broker Bean*, a relationship between the first identified *Broker Bean* graph node (BB1) and the one identified in the variable data type (BB2) should be created as “BB1 composed\_of BB2”.** Since the *Broker Beans* represent the business entities flowing in the system and, in some cases, a business entity can be composed of other business entities, this relationship should be identified.

**DR7: In the *IEB* components' files, a relationship between graph nodes should be created when Inheritance or Implementation relationships are identified.** For some components (e.g., *Producers* or *Consumers*), the collection of data representing the Inheritance and Implementation between the files that define them is relevant and must be ensured.

**DR8: For any system component that instantiates another one, a relationship between the component (Cp1) and the one identified as being instantiated (Cp2) should be created as “Cp1 instantiates Cp2”.** This step will ensure that all the relationships between the components are stored to generate knowledge about the interaction of the components.

**DR9:** For the *Consumers*, it should be identified if they verify certain *Rules*, creating a relationship between the *Consumer* (C1) and the *Rules Session* that is executed (RS1) as “C1 runs RS1” and which *Rule* (Ru1) is verified as “C1 verifies Ru1”. With this design rule, it is guaranteed that the *Consumers* running the *Rules Session* (a set of *Rules* within all the defined *Rules*) and therefore verifying certain *Rules*, are identified, and properly stored in the graph.

**DR10:** For each *Consumer*, it should be identified if it queries or stores data in the *Event Aggregator* component. A relationship between the *Consumer* (C1) and the used *Event Aggregator* (EA1) should be created as “C1 queries EA1”. Moreover, a relationship between the *Consumer* and the *Event Aggregator* where it stores new data (EA2) should be created as “C1 stores\_data\_in EA2”. This design rule identifies an interaction between a *Consumer* and the *Event Aggregator* in both directions: querying and storing data on it.

**DR11:** For the identified *Triggers*, it should be created a relationship between the *Trigger* (T1) and the *Destination System* (DS1) that will receive the data sent by the *Trigger* as “T1 propagates\_data\_to DS1”. The actions defined in the *Triggers* can send data to different *Destination Systems* identified in the system architecture presented in [1], the reason why each *Trigger* should have a relationship with the graph node that represents the *Destination System* being used in that action.

**DR12:** Regarding the rules’ repository, it should be collected the *Rule*’s name and the *Trigger* fired by the *Rule*, as well as the *Broker Beans* used for the *Rule* verification and to trigger the action. The *Rule*’s names should be saved as graph name nodes and the *Trigger* and *Broker Beans* identified are used for design rules DR13 and DR14. Depending on the *Rules Engine* being used, different ways for the rules definition can exist (i.e., Drools as a way of defining rules, while other business rules systems may have others). Nevertheless, the *Rules* must be identified and stored in the graph, as well as the *Triggers* and the *Broker Beans* used by them.

**DR13:** For the *Rules* identified in DR12, it should be created a relationship between the *Rule* graph node (Ru1) and the *Broker Bean* (BB1) used for the *Rule* verification as “Ru1 uses BB1”. Moreover, it should be created a relationship between the *Rule* and the graph node that represents the *Trigger* (T1) fired by the *Rule* as “Ru1 fires T1”. These relationships will ensure the tracking of which *Broker Beans* are used by the *Rules* verification and which *Triggers* are fired by which *Rule*.

**DR14:** For the *Triggers* identified in DR12, it should be created a relationship between the *Trigger* graph node (T1) and the *Broker Beans* (BB1) used to take any action as “T1 uses\_to\_trigger BB1”. It is necessary to identify which specific *Broker Beans* are used for the system’s actions ensuring the identification of the data that are propagated to the *Destination Systems* (e.g., IoT Gateways or a database to feed Analytical Applications, as identified in [1]).

## 4 Demonstration Case

In this section, it is presented a demonstration case for the *Code Inspector* component of the architecture proposed in Sect. 3. First, it is presented a flowchart that reflects the implementation of the set of design rules defined in Subsect. 3.2, using the *IEB*

implementation in the Bosch Portugal ALR<sup>8</sup> data context. Then, for the obtained flowchart, a part of the GDM is demonstrated and explained, resulting from the implementation of the steps in the flowchart. Although the *Logger* and *Logs Processor* components are already being developed, for this paper, this component is defined at the conceptual level, and its demonstration is identified as future work.

#### 4.1 Code Inspector Flowchart

The flowchart (Fig. 2) that represents the implementation of the design rules defined in Subject. 3.2 is based on the *IEB* system already presented in [1]. To properly interpret this diagram, remember that the *IEB* was implemented using Java and Drools (this last one, as Rules Engine).

In this context, each package represents the implementation of one of the *IEB* components and to save this data, the *Code Inspector* seeks throughout the packages to analyze the implemented code, storing the package name as 1<sup>st</sup> level label (**DR1**). When inside a package, it searches for *.java* and *.drl* files (Drools files) and for each file, its directory name is saved (if it does not exist yet) as 2<sup>nd</sup> level label (**DR2**). After that, the file name is saved as the graph node, representing the class that is part of the system implementation (**DR3**). The identified labels are then linked to the created node (**DR4**). All these steps are executed until there are no more packages and no more *.java* or *.drl* files to identify. This will guarantee that the graph nodes and labels needed for the creation of the relationships already exist.

With all the graph nodes already created, the *Code Inspector* starts exploring the files again in the first package. If the selected file is from the *Broker Beans* package, the private variables names and properties are saved as node properties for the graph node previously defined with the same name as the file name being analyzed (**DR5**). If some of the variable's types represent other nodes identified in the *Broker Beans* package, a relationship between the file/graph node being analyzed and the graph node representing the variable type is saved as shown in Fig. 2 (**DR6** - *composed\_of* relationship).

For the rest of the *.java* files identified in the packages, if they include a string “*extends*” or “*implements*” followed by another graph name node previously identified, a relationship between the file/graph node being analyzed and the one identified after the mentioned strings are saved as identified in the flowchart (**DR7** - *extends* and *implements* relationships). The string “*new*” followed by another graph name node previously identified will allow the identification of instantiations being carried out by the file under analysis. A relationship is saved as a file/graph node that *instantiates* another graph node (**DR8**). The same happens for the identification of which *Consumers* run the *RulesSession*. In this case, searching for the string “*getStatelessSession*” and saving a relationship as the graph node representing the file being analyzed *runs* the “*ruleSessionName*” (the *getStatelessSession* parameter) identified.

Furthermore, the type of the parameter from the “*executeRules*” method will allow the identification of the *Rules* verified in that session (all the *Rules* waiting for a

<sup>8</sup> A system that verifies if a lot can be shipped to the customer.



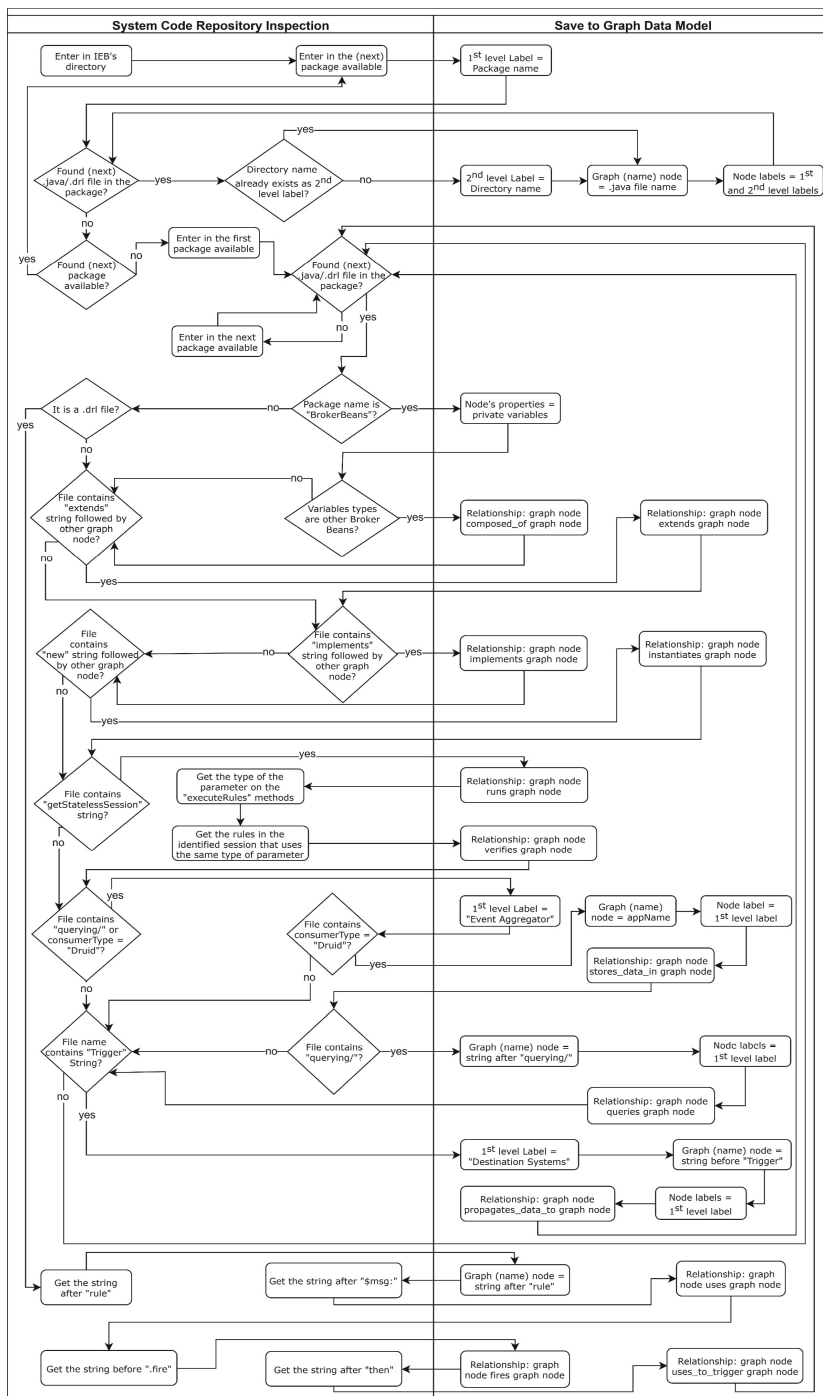


Fig. 2. Application of the design rules in the IEB

specific parameter type inside a session will be verified by the *Consumer - DR9*). Regarding the *Event Aggregator* component, this one can be queried, and the data can be stored on it. If the file contains the string “*querying/*” or the “*consumerType*” equals “*Druid*”, *Event Aggregator* is saved as 1<sup>st</sup> level label. For the ones in which the “*consumerType*” equals “*Druid*”: i) the “*appName*” (that represents the *Event Aggregator*) is saved as graph name node; ii) the 1<sup>st</sup> level label is defined as a label for the created node; and, iii) a relationship is created as the node representing the file being analyzed *stores\_data\_in* node representing the *Event Aggregator*.

The same happens when the “*querying/*” string is found, being the relationship defined as *queries* instead of *stores\_data\_in* (**DR10**). On the other hand, if the file name contains the “*Trigger*” string: i) *Destination Systems* is saved as 1<sup>st</sup> level label; ii) the name before “*Trigger*” is saved as graph name node; iii) the 1<sup>st</sup> level label is defined for the created node; and, iv) a relationship is created between the node representing the file being analyzed (*Trigger*) and the node created as being the *Destination System* (“*Trigger propagates\_data\_to Destination System*”) (**DR11**).

Concerning the *Rules* defined in *.drl* files (Drools files), a file exploration process is executed to collect: i) the *Rules* that are in quotes after the “*rule*” string; ii) the *Broker Beans* used for the verification of the rules; iii) the *Trigger* to be fired by the rule; and, iv) the *Broker Bean* to be fired by the *Trigger*. With this data, are created: i) the graph node with the *Rule* name (**DR12**); ii) a relationship between the *Rule* graph node and the *Broker Bean* used for its verification (**DR13**); iii) a relationship between the *Rule* graph node and the *Trigger* that is fired (**DR13**); and, iv) a relationship between the *Trigger* and the *Broker Bean* that was flowing when the *Trigger* was fired (**DR14**).

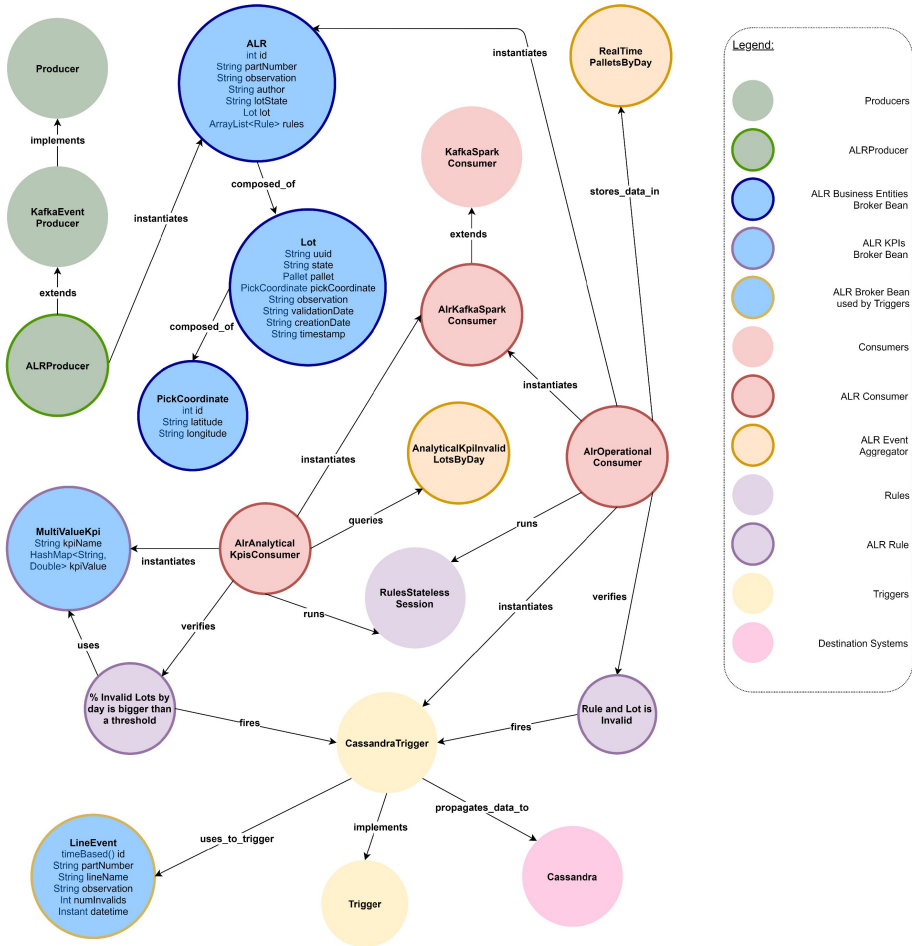
With the definition of the flowchart, it is possible to understand that the design rules presented in Subsect. 3.2 are easily transformed into small tasks to be coded and applied to the implemented system. Although the system is implemented using Java and Drools, other technologies can be used if the main guidelines are followed, as the application of the design rules returns the data needed to be monitored.

## 4.2 Graph Data Model

In Fig. 3, it can be seen the representation of the GDM that was obtained from the *Code Inspector* component to support the *Mapping and Drill-down System* (see Subsect. 3.2). Due to the difficulty of presenting here the whole GDM created during this work, this figure only shows an example of the relationships between the possible types of graph nodes. Nevertheless, the whole data model contains sixteen labels, more than fifty nodes and more than sixty relationships.

Before starting the explanation of the nodes (circles) and relationships (edges) of the GDM in Fig. 3, it should be considered that the nodes’ backgrounds represent the packages’ name (1<sup>st</sup> level label mentioned in **DR1**), and the lines around the nodes represent the folders’ names (2<sup>nd</sup> level label mentioned in **DR2**) as shown in the figure legend. **DR3** and **DR4** are explicit in the GDM since the nodes are clearly identified and all of them have a specific colour.

Considering this, the different types of graph nodes and relationships between them are described as follow:



**Fig. 3.** GDM resulting from the application of the set of proposed design rules (Color figure online)

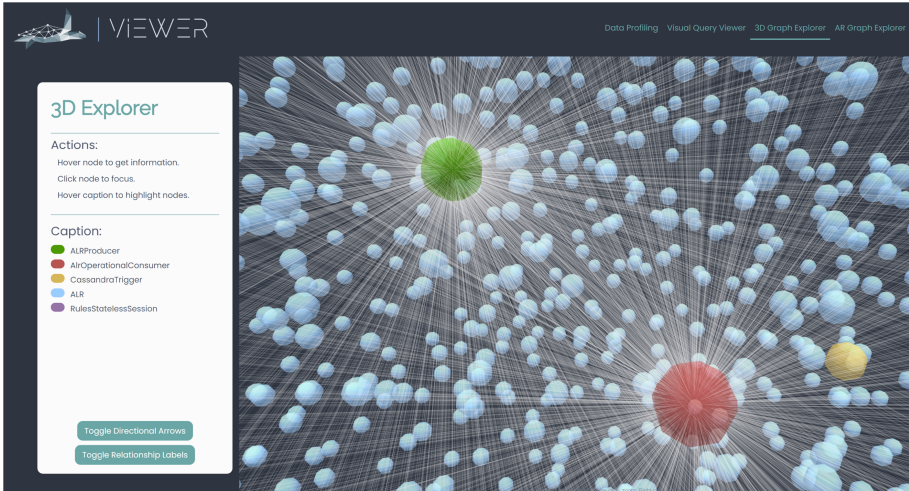
- The *ALRProducer* extends a generic implementation of a producer developed using Apache Kafka (*KafkaEventProducer*), which by itself implements the *Producer* interface (**DR7**). This aims to standardize the implementation of the *Producers*, creating a generic class and an interface that reflects the behaviour of every producer in the system.
- Three types of *Broker Beans* can be seen: i) *Broker Beans* representing the business entities that arrive at the system through the events (*ALR*, *Lot* and *PickCoordinate*); ii) *Broker Beans* representing the KPIs calculated by the system (*MultiValueKpi*); and, iii) *Broker Beans* created during the verification of the *Rules* to be later used by the *Triggers* (*LineEvent*). Regarding the *Broker Beans* that represent the business entities, these are instantiated by *Producers* and *Consumers*

(DR8). They can also be composed of other *Broker Beans* (an *ALR Broker Bean* is composed of a *Lot Broker Bean* which subsequently is composed of a *PickCoordinate Broker Bean* - DR6). The KPIs *Broker Beans* (*MultiValueKpi*) are instantiated by *Consumers* with analytical purposes (*AlrAnalyticalKpisConsumer*) (DR8) and they are used by the *Rules* dedicated to the same analytical goals (*% Invalids Lots by day is bigger than a threshold*) (DR13). The third type of *Broker Beans* (*LineEvent*) is used by the *Triggers* that propagate them to the *Destination Systems* (DR14). For every *Broker Bean*, their variables and data types are identified as node's properties (DR5).

- The operational or analytical *Consumers* (*AlrOperationalConsumer* and *AlrAnalyticalKpisConsumer*) instantiate (DR8) a specific class (*AlrKafkaSparkConsumer*) that extends the *KafkaSparkConsumer* (DR7), being this a specific code design strategy in the *IEB*. Both of them run the *RulesStatelessSession* and verify the defined *Rules* in the GDM (DR9). Furthermore, *Consumers* have two types of interactions with the *Event Aggregator* presented in the GDM (e.g., relationship *stores\_data\_in* between the *AlrOperationalConsumer* and the *RealTimePalletsByDay*, and relationship *queries* between the *AlrAnalyticalKpisConsumer* and the *AnalyticalKpiInvalidLotsByDay* - DR10). Finally, *Consumers* instantiate the *Triggers* (e.g., *CassandraTrigger*) that can be fired after the verification of the *Rules* (DR8).
- Two *Rules* are defined in the GDM (*% Invalids Lots by day is bigger than a threshold* and *Rule and Lot is Invalid* - DR12) with a relationship to the *Trigger* node reflecting that a *Rule* fires a *Trigger* (DR13).
- Regarding the *Triggers* component, the *CassandraTrigger* implements the *Trigger* interface (DR7) and *propagates\_data\_to* the *Cassandra Destination System* (DR11).

The detailed GDM (Fig. 3) as well as the design rules identified in the description of the GDM, reflect the successful application of the design rules in the *IEB* system being demonstrated in the Industry 4.0 movement of Bosch Portugal. The codebase of the system was thoroughly analyzed by the authors, comparing it to the generated GDM. With this validation, it was possible to conclude that, as an initial prototype, the data needed to monitor the continuous growth of the system (e.g., new *Producers* or *Consumers*) is adequately identified and captured by the design rules proposed.

Regarding the *Logger* component, the experimental work in progress is currently focused on logging the execution of the *ALRProducer* and the *AlrOperationalConsumer*. The logs are being stored and processed as described in Sect. 3 and the *Web Visualization Platform* [2] is used (with a new design) for the data analysis. In the analysis presented in Fig. 4, it can be seen the *ALRProducer*, the *AlrOperationalConsumer* and the *CassandraTrigger* nodes (green, red and yellow nodes) connected to all the *ALR Broker Beans* nodes (blue). In the future, we will work on extending the GDM presented here with the processed logging information from the *IEB* system.



**Fig. 4.** Data from the *Logger* component presented in the 3D graph explorer of the IEB Web visualization Platform (Color figure online)

## 5 Conclusions

Given the potential complexity of the *IEB* running in industrial contexts, we consider that adequate monitoring of this system is essential. This monitoring was conceptually considered in the *IEB* system [1] with the proposal of the ***Mapping and Drill-down System*** supported by graph storage and analysis technologies.

In this paper, the architecture for the whole feeding system that will fuel the ***Graph Database*** was presented and discussed considering two main parts: i) the ***Code Inspector***; and, ii) the ***Logger***. This feeding system is responsible for generating the ever-growing graph mentioned above, addressing innovative monitoring capabilities for a CEP in Big Data contexts.

For the ***Code Inspector*** component, the main focus of this work, a set of design rules was defined to ensure that the *IEB* components are continuously and automatically inferred from the codebase, as well the relationships between them. The design rules were applied in a Bosch Portugal demonstration case using the ALR data, showing a flowchart on how to properly inspect the *IEB* codebase, and highlighting the results in a detailed GDM. The design rules were successfully applied to the *IEB* system returning all the useful graph data (labels, nodes, relationships) that is needed to feed the ***IEB Mapping and Drill-down System***.

Despite the focus on the *IEB* system being demonstrated in the Bosch Portugal context, we believe that the artefacts and insights provided here, which complement the ones in [1, 2], are conceptually, technically and technologically relevant for several researchers and practitioners in the area. No other system similar to the *IEB* considers the relevance of monitoring using consistent strategies that focus on the evolution and growth of the system when working in production contexts, being this a key point to guarantee the adequate maintenance of the system in many contexts like the industry 4.0 movement.

**Acknowledgements.** This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020, the Doctoral scholarship PD/BDE/135101/2017 and by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project n° 039479; Funding Reference: POCI-01-0247-FEDER-039479].

## References

1. Andrade, C., Correia, J., Costa, C., Santos, M.Y.: Intelligent event broker: a complex event processing system in big data contexts. In: AMCIS 2019 Proceedings. Cancun (2019)
2. Rebelo, J., Andrade, C., Costa, C., Santos, M.Y.: An Immersive web visualization platform for a big data context in bosch's industry 4.0 movement. Presented at the european, mediterranean and middle eastern conference on information systems (EMCIS), Dubai, Dec 2019
3. Hadar, E.: BIDCEP: A vision of big data complex event processing for near real time data streaming. In: CAiSE Industry Track (2016)
4. Flouris, I., Manikaki, V., Giatrakos, N., Deligiannakis, A., Garofalakis, M., Mock, M., et al.: FERARI: a prototype for complex event processing over streaming multi-cloud platforms. In: Proceedings of the 2016 International Conference on Management of Data. pp. 2093–2096. ACM, New York (2016)
5. Jha, S., Jha, M., O'Brien, L., Singh, P.K.: Architecture for complex event processing using open source technologies. In: 2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE), pp. 218–225 (2016)
6. Nguyen, F., Pitner, T.: Information system monitoring and notifications using complex event processing. In: Proceedings of the Fifth Balkan Conference in Informatics. pp. 211–216. Association for Computing Machinery, Novi Sad, Serbia (2012)
7. Jayan, K., Rajan, A.K.: Sys-log classifier for Complex Event Processing system in network security. In: 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 2031–2035 (2014)
8. Nguyen, H.T.C., Lee, S., Kim, J., Ko, J., Comuzzi, M.: Autoencoders for improving quality of process event logs. *Expert Syst. Appl.* **131**, 132–147 (2019)
9. Gupta, S.: Introduction to Application Logging. In: Gupta, S. (ed.) *Logging in Java with the JDK 1.4 Logging API and Apache log4j*, pp. 1–9. Apress, Berkeley (2003)
10. Ćurin, T., Bogadi, D., Volarević, M., Štajcer, M., Mihalić, A., Mock, M.: Final Application Scenarios and Description of Test Environment. Hrvatski Telekom (2016)
11. Dickey, D.A., Dorter, B.S., German, J.M., Madore, B.D., Piper, M.W., Zenarosa, G.L.: Evaluating Java PathFinder on Log4J. vol. 15, Carnegie Mellon University (2011)