Ontological Approach to Automated Analysis of Enterprise Data Storage Systems Log Files

Olga Mamoutova, Mikhail Uspenskiy, Sergey Smirnov, Marina Bolsunovskaya

Peter the Great St. Petersburg Polytechnic University, Polytechnicheskaya st. 29, 194021 St. Petersburg, Russia; {olga.mamoutova, mikhail.uspenskiy, sergey.smirnov, marina.bolsunovskaia}@spbpu.com

Abstract: Enterprise data storage is usually designed to operate as a highly available system, which needs continuous monitoring and diagnosing of a system state. However, applying the traditional approach to administration tasks with manual analysis of event log files is infeasible due to the complexity of such systems. Multiple levels of monitoring and the heterogeneous nature of diagnostic data require an autonomous solution that provides a combination of model-based, knowledge-based and data-based approaches. An ontology-based diagnostic model, that integrates an expert knowledge of diagnostic parameters, typical storage configurations, and common failure modes, can be considered a promising solution for this task. The implementation goal for such an autonomous diagnostic approach would be not to substitute, but to complement existing diagnostic infrastructure. Hence, software and system event log files can be viewed as additional diagnostic data to be analyzed. This paper presents a new approach to event log analysis, which is supported by the ontology-based diagnostic model: structure of supporting ontology classes, text preparation algorithm, key implementation points, and assessments of the data mining algorithm suitability for the task.

Keywords: data storage system; diagnostic model; knowledge-based systems; ontology; event log; data mining

1 Introduction

Companies, which use data storage systems for business-sensitive data, require seamless service delivery and have zero tolerance for data loss, loss of service, and performance degradation [1]. A reason for a data storage system failure can be a hardware component fault, software error, or misconfiguration. As a result, a data storage system usually employs a combination of redundancy techniques and implements a health and fault management strategy to detect and diagnose developing issues and provide access to the data in presence of a possible failure [2].

A typical configuration of an enterprise data storage system includes several enclosures with drives to store data, a cluster of storage controllers to manage user requests and perform service functions, and network infrastructure to connect users, storage controllers and drives [3]. To provide the necessary diagnostic data, continuous monitoring must be performed at all levels of a data storage system. The lowest level of monitoring gets diagnostic data from terminal sources with no data regarding causes of reported states: built-in self-test functions or embedded sensors data. Some examples of such data are S.M.A.R.T. or SCSI diagnostic data for disks, signal quality for a network transceiver, fan speed, and temperature readings. For the next, system-level, monitoring, a hierarchy of distributed tools gathers monitoring data across hardware and software components of the lowest level to get a full operational picture of the system: components health reports, ambient parameters, network performance readings, as well as metrics of storage and computing nodes workload, performance, capacity, and power consumption. Finally, a file or block-level performs service-level monitoring across the corresponding physical, virtual and, if present, cloud components.

Such health and performance monitoring is usually executed by a combination of vendor-specific and third-party software. Monitoring tools gather data by polling status data and recording alarms, errors, and miscellaneous messages. Machine-learning techniques for anomaly detection can provide additional data for analysis. Further diagnostic tasks include error and event correlation, Root Cause Analysis (RCA), and deciding on necessary management procedures. Because of the heterogeneous nature of a diagnostic data and because these diagnostic tasks require a detailed understanding of a system architecture, field experience, and best practices knowledge, diagnostics are primarily performed by an administrator [3]. As a result, monitoring data are often stored in the form of human-readable log files. Section 2 gives a review of current monitoring and diagnostic practices.

The embedded ontological approach is a way towards automation of the diagnostic procedures in a data storage system. Applying autonomous monitoring and embedded diagnostics as part of a self-management strategy for a storage processor results in a faster real-time fault detection as opposed to reactive administrator intervention. When using the presented ontological approach, a hybrid diagnostic model of a particular system is created, including an unstructured heterogeneous expert knowledge in a formalized form, an explicit white-box like reliability model of the data storage system and a black-box-like model in the form of pre-trained machine learning algorithms. Section 3 contains the key details regarding the implementation of the ontological approach. Section 4 contains the details regarding the application of the approach with the traditional event log data.

The scientific contribution of the presented research is the ontological approach that allows to create a sophisticated, but flexible and easily upgradable diagnostic solutions for data storage systems despite a wide range of possible faults of varying origin and importance. This paper extends the work originally presented

in the 17th IEEE International Symposium on Intelligent Systems and Informatics [4] and aims to provide a thorough description of the suggested ontological approach, methods to define symptoms, and its application in case of event log files as monitoring data. It provides results and recommendations based on practical evaluation of the approach for a medium-sized enterprise data storage system.

2 Related Work

2.1 Methods of Data Analysis for Data Storage Diagnostics

Identifying occurring or emerging problems in the form of an event or performance deviation is the first step of diagnosing, which needs the details of logical or physical entities of a system. Storage resource management tools [5] provide an administrator of a data storage system with necessary remote monitoring and diagnosis services: dashboarding, thresholding, and custom policy configuration [6]. Some tools additionally perform data-driven anomaly detection [7-9].

The next step is RCA to diagnose the cause of the detected problem, which allows deciding on further management actions. One way to perform RCA is to manually trace down the history of events in error and configuration log files. A monitoring tool can assist in this task by mapping the detected problem to the network topology, searching for event correlations, and showing the related resources. However, for networks alone, RCA is a challenging task [10], and adding hosts, power supply and storage devices to the systems only adds to its complexity, resulting in hundreds of event types that demand administrator attention. Hence, manual RCA becomes impractical.

An example of automated RCA is IPASS algorithm for Storage Area Networks (SAN) [11], which reconstructs I/O paths and performs an informed search for the root cause of a performance problem. IPASS requires a graph representation of a SAN, which could not be reused as a diagnostic model for another SAN or other data storage system. Another limitation is that IPASS uses only performance metrics and does not account for the health or capacity metrics of the network nodes.

Another example is RCA implemented for IT environments in eG Enterprise [12]. This RCA algorithm discovers dependencies and connections between system components with corresponding health metrics, then creates pairs of relational maps and dataflow graphs to describe them and searches for correlations among connected or dependent problematic components. This approach has a visibility

limitation as it operates at the application level and does not take into account the internal structure of the devices, which limits its insight into problems.

The new approach, presented in this paper, aims to solve the problem of automatic RCA by implementing an embedded service that runs on a storage processor. Such embedded diagnostic leverages visibility of internal organization of a data storage system and allows a storage processor to perform real-time diagnosis and self-management. Furthermore, the new approach uses an ontology to solve the problem of constructing a diagnostic model for arbitrary data storage systems with various redundancy schemes and types of monitoring data and also to introduce heterogeneous expert knowledge into the model.

2.2 Ontologies for Diagnostics

In computer science, ontology is a type of knowledge representation model that is able to formally specify a shared knowledge in a certain subject area. Concepts, objects, and relations between them can be described by a hierarchy of classes and individuals, their attributes, and object and data properties [13].

Ontologies are widely used in the field of technical diagnostics. They provide a unified terminology between communicating distributed diagnostic agents [14], replace a standard way to describe configurations of industrial equipment [15], and describe communicating components in the system [16]. Another popular approach is to augment diagnostic models with ontologies to verify sensor data [16-18].

There are approaches, in which an ontology serves as a core of a diagnostic model. However, they are designed for other knowledge domains: mainly the automotive domain [19-21], but also for steam turbines [22] and ventilation systems [23] – and are not applicable for data storage system diagnosis. For example, Saeed et al. [24] describe a general ontological approach to the active diagnosis of sensors and actuators in embedded systems.

Schoenfisch et al. [25] proposed to perform RCA for IT infrastructures by reasoning over a Markov logic network, obtained from an ontology that models a system being diagnosed. The ontology defines types of components and possible relations between them, including a dependency graph. However, the ontology includes the information only regarding the potential risks for the components, therefore, it cannot be used for fault detection purposes. Moreover, the approach is centered around a user, who constructs a model and performs RCA, which limits its applicability for autonomous embedded diagnosis.

Similarly, to work by Dendani et al. [26], the presented approach resembles a knowledge-based system within a case-based reasoning methodology [27], namely, its case representation and case matching and retrieval aspects. A new ontology at the core of the presented approach aims to support all steps of

diagnosis for a data storage system: monitoring, fault detection, RCA, and deciding on management actions.

2.3 Software Logs as a Source of Diagnostic Data

A typical enterprise data storage system contains multiple server-based nodes, such as storage controllers, fabric controllers, and other components with a sophisticated software ecosystem on top of these hardware components. The software includes operation systems, security and authentication services, data managing services, clustering services, hardware monitoring, and management services. Most of them provide some kind of logging features and, as a result, a typical data storage system generates large volumes of different software logs.

According to their source type, software logs can be classified [28, 29] into event logs, application logs, service logs, and system logs (or into operation system logs, service logs, and application logs as in [30]). System and event logs, such as "dmesg", "faillog" and "messages", are the most obvious sources of diagnostic data, but other types of logs contain useful information as well.

Although the aforementioned logs can be exceptionally important as a source of knowledge about the system's health, there are some difficulties that make them hard to use in enterprise products. Perhaps the main issue, as stated by Zhu [31], is an excessive amount of details in software logs: a typical application log contains a lot of data not related to any particular issue or error. Because of that, during the process of identifying the root cause of a fault, administration staff is forced to manually analyze hundreds of megabytes (or even gigabytes) of event logs, which makes such a procedure extremely time-consuming.

A possible solution to this problem is automated log mining. The most common approach to automated log analysis relies on the interpretation of a log as a structured chain of consecutive events and on log template extraction (see Pande et al. [32], He et al. [33], and Hamooni et al. [34]). These methods usually require some a priori knowledge about the log structure, log message's structure, reasons, and sources of different message types. Hence, some degree of manual analysis of each log type, used as a source of diagnostic data, is still needed.

Another possible approach, presented for example by Bertero et al. [35], handles a software log as raw text data, which can be analyzed using natural language processing methods. As a result, each log type can be used as a source of diagnostic data without prior study. Because of that, although these methods provide less accurate results compared to the methods, based on a log structure analysis, they are more preferable in case of an implementation coupled with an ontological diagnostic model.

3 Ontological Approach to Data Storage System Diagnostics

3.1 Ontology of Data Storage System Health

A set of root disjoint classes and a set of property types as an ontology schema are the basis for the presented diagnostic approach. Each root class has a tree of subclasses that represents related concepts in the data storage reliability domain: for example, subclasses from the *Parameter* tree are the names of known types of monitoring data as diagnostic parameters, subclasses from the *SystemFault* tree are the names of known types of system events. Object properties describe relationships between particular concepts: for example, a relationship *may_lead_to* between two subclasses from the *ComponentFault* indicates that one particular type of fault is known to lead to another type of fault. Figure 1 shows all root classes of the schema and types of object properties that can be used to describe relationships between pairs of subclasses. Data and annotation properties provide specific characteristics of separate concepts to support further implementation of diagnostic procedures.

The target users of the ontology are developers of the diagnostic system, who can be considered experts in the field. They may extend existing core ontology by adding subclasses and object and data properties to describe specific knowledge of data storage systems. Then, in order to represent a hierarchical model for a data storage system being diagnosed, that is declared by a *StorageSystem* individual, they should create individuals of subclasses from *Component* and *StorageSubsystem* trees connected by *consists_of* object properties (see Fig. 1).

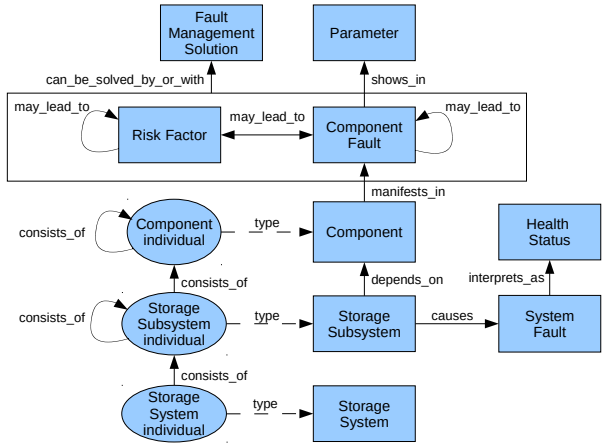

Figure 1

Schema of data storage health ontology

A particular knowledge base derived from this schema should cover the following competencies:

- system configuration competency: what is the current configuration of the data storage system and what are the names of its components and subsystems (*consists_of* object properties);
- symptom competency: which monitoring data may be available and how to identify faults based on the monitoring data (*shows_in_parameter* object properties, also data and annotation properties for subclasses from *Parameter* and *ComponentFault*);
- component operational status competency: how to identify a component's level of performance based on the detected faults (*if_fatal_manifests_in* and *if_warning_manifests_in* object properties);
- subsystem reliability function competency: how a subsystem's operational status depends on its components' statuses (*strongly_depends_on*, *majorly_depends_on*, and *depends_with_ecc_on* object properties);
- root cause competency: what may have caused the fault and which system-level events are caused by the detected faults (*may_lead_to* object properties, also *if_warning_causes* and *if_failed_causes* object properties);
- system health competency: what is the severity level of a system-level event (*interprets_as* object properties);
- management competency (optional): which fault containment procedures can be recommended for the detected faults (*can_be_solved_by_or_with* object properties);
- source information competency (optional): what are the synonyms and translations of specific terms and which sources of information can be referenced (various annotation and data properties).

Besides the support of diagnostics, this ontology can also be used for fault injection during system reliability validation and for a preliminary reliability evaluation at early design stages.

The verification plan for the ontology includes formal coherency and consistency check, checks on compliance with common ontology design rules (with OntOlogy Pitfall Scanner![1]), unit tests, and crowd-sourcing validation. Custom unit tests capture the following requirement: for every individual associated with the current system configuration, the ontology has to have a full path of related object properties to cover all the ontology competencies. Otherwise, a diagnostic model of the system is not complete. On the other hand, if ontology contains a full path for some concept, like a component type or a diagnostic parameter, but the fault

[1] Can be found at http://oops.linkeddata.es/

and health management architecture does not include a corresponding entity, a review of that architecture may be initiated.

The ontology is implemented in Protégé software [36], which is the most often used ontology-editing tool for scientific and academic projects. Currently, the core knowledge base in the form of ontology contains around 2300 axioms and over 500 classes to describe reliability aspects of a data storage system with domain-specific standards, articles, application notes, and in-house expert knowledge as sources of information. The collection of classes includes various components (storage, network, and processing devices, software services), subsystems (storage pool and volume, cluster, network, and other elements of control and data paths), faults (faults by component type, risk factors as environmental factors, human errors or non-monitorable component faults and system faults by subsystem type), extensive set of monitoring parameters (including SMART and SCSI log parameters for disks and typical network interface parameters) and fault management solutions (primarily for network-related faults) [2].

3.2 Knowledge Base Implementation

The recommended approach to implement a particular diagnostic system is to convert a knowledge base from a human-friendly RDF/XML format of an ontology to the N-Quads format of a graph database, which is more appropriate for further machine interpretation. The alternative currently used approaches employ Java frameworks and have limited applicability in the case of embedded implementation. A graph database of choice would be Dgraph (v1.1): compared with alternatives it has an outstanding performance level, is open-source with Apache Public License 2.0, uses Go as a native language and implements GraphQL+– query language[3].

Figure 2 shows the overview of the presented approach. Experts use Protégé as an ontology editor to update the data storage health ontology. Further knowledge manipulations are supposed to be executed by services embedded in a storage controller of a data storage system: custom reporting service provides a system configuration and current monitoring data; a knowledge base builder service converts an ontology to a graph database and supplements it with a system configuration data; and diagnostic service performs the necessary queries to a graph database in order to execute a diagnostic algorithm. Obtained fault detection results and additional interpretation information should be forwarded by a storage processor to an administrator of the system.

[2] Can be found at https://github.com/Mamoutova/data-storage-diagnostic.

[3] Documentation on GraphQL+– can be found at https://docs.dgraph.io/query-language.

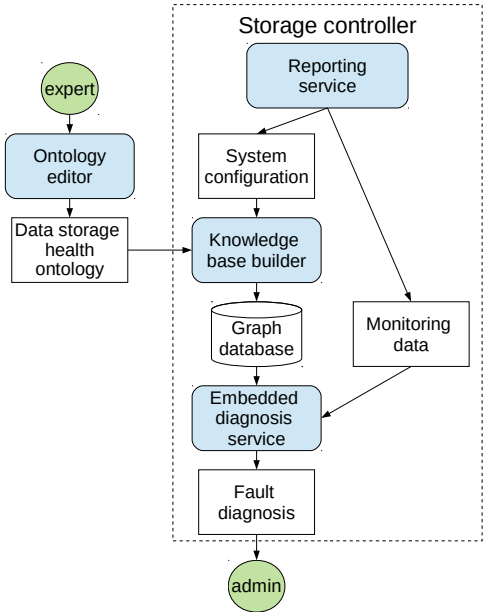

Figure 2
Key elements of the ontological approach to data storage fault diagnosis

To transform an ontology into a graph database, classes and individuals should be converted to graph nodes, while data and object properties – to graph edges. The hierarchical nature of properties can be translated to facets, which allow annotating edges of a graph with arbitrary information.

However, for object properties, the proposed conversion cannot be made directly. According to the Web Ontology Language (OWL) specification, object properties must connect pairs of individuals. On the contrary, the ontology of data storage system health heavily utilizes object properties to connect pairs of subclasses. The latter is properly supported by Protégé as an ontology editor though, with auxiliary *owl:Restriction* classes. Table 1 shows an example description of *if_fatal_manifests_in* object property connecting *Component* and *BadHealth* in the RDF/XML format and a desired corresponding description in the N-Quads format.

Table 1
Example definition of an object property in the RDF/XML and N-Quads formats

Format	Description example

RDF/XML	`<owl:Class rdf:about="urn:#`**`Component`**`">`
	`<rdfs:subClassOf>`
	`<owl:Restriction>`
	`<owl:onProperty`
	`rdf:resource="urn:#`**`if_fatal_manifests_in`**`"/>`

	<owl:someValuesFrom rdf:resource="urn:#**BadHealth**"/> </owl:Restriction> </rdfs:subClassOf> </owl:Class>
N-Quads	<urn:#**Component**><urn:#**if_fatal_manifests_in**><urn:#**BadHealth**>.

This simplified schema of a graph database allows streamlining the further implementation of a diagnostic algorithm. Table 2 shows an example of a query to the ontology and a corresponding query to the graph database: both queries search for the name of a system event resulting from the transition of a storage pool to the health level “warning”.

Table 2

Queries to knowledge base in the form of ontology (SPARQL language) and in the form of graph database (GraphQL+–)

Query language	Query example
SPARQL (ontology)	SELECT ?event WHERE{ ?entity rdfs:label "StoragePool_id" . ?entity rdfs:subClassOf ?aux . ?aux owl:onProperty ?property . ?property rdfs:label "if_warning_causes" . ?aux owl:someValuesFrom ? event. }

GraphQL+– (graph database)	status (func: uid(StoragePool_uid)) {
	causes
	@facets(eq(rank, if_warning_causes)) {
	name }

Besides simplified queries, this transition from the ontology format to the graph database format allows accelerated machine processing of queries to the knowledge base due to better inherent data indexing. As a result, higher performance of the diagnostic algorithm enables its easier real-time implementation in presence of large amounts of monitoring data.

3.3 Diagnostic Algorithm

The proposed diagnostic algorithm has a modular structure: it uses component operational status and symptom competencies of the ontology to evaluate a component state based on current monitoring data; then it uses system configuration, subsystem reliability function, and root cause competencies to find relevant subsystems and evaluate their states; finally, it uses system health competency to report overall health of a data storage system. Thus, a particular diagnostic procedure can be reconfigured by choosing an appropriate trigger and

order of the function execution: by monitoring changes in diagnostic parameters with a bottom-up approach, or by a massive periodic status check of all subsystems and components with a top-down approach.

The necessary preparation step for the diagnostic algorithm includes the actualization of a system configuration followed by ontology verification. Further execution of the algorithm can be performed by a storage processor in a real-time manner.

For a bottom-up approach, a change in monitoring data of a component triggers algorithm execution. The fault diagnosis algorithm looks like the following, in which every step can be implemented with a corresponding GraphQL+– query:

1) Identify the type of the component and all parent component types.
2) Search for component faults associated with the tree of component types.
3) Search for the fault symptoms.
4) Check whether the current monitoring data confirm any of the symptoms.
5) Evaluate the component health state.
6) Recursively search through the system configuration for the subsystems that include the component and other subsystems that include the found subsystems. For every found subsystem reconstruct its reliability function and evaluate its health state.
8) Identify the names of the system events corresponding to the tuple of subsystem health levels.
9) Interpret the tuple of system events in terms of the overall data storage system health.
10) Search for the associated faults and risk factors for the tuple of detected component faults and system events and search for the corresponding recommended fault containment procedures.

For example, suppose that one of the disks in a system reports an increasing number of corrected errors while having over a year of accumulated power-on hours, which also shows in background scan results. According to the expert knowledge, this indicates an expected degradation of the disk media, which can be interpreted as a warning state of the disk (see Fig. 3). Suppose that the disk is in a storage pool with a 10+2 error correction scheme. When two disks get a warning state, the state of the pool becomes vulnerable. Hence, the states of other storage pools that include this pool and the state of the whole data path become vulnerable as well.

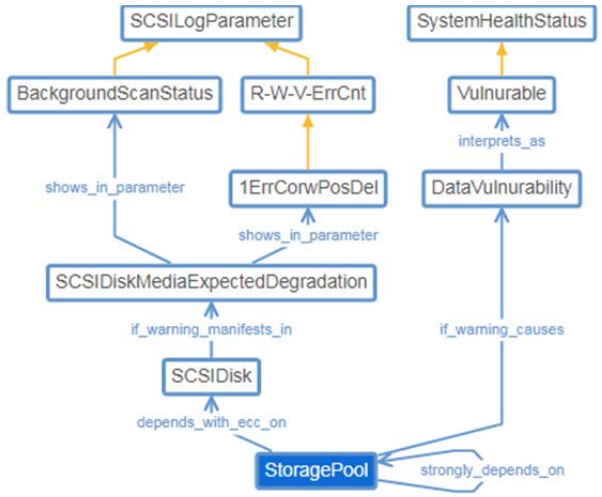

Figure 3
Segment of the ontology of data storage system health regarding health of a storage pool

3.4 Three Types of Symptom Definition

The key to successful fault detection is a thorough description of fault symptoms together with continuous monitoring of corresponding diagnostic parameters. To cover the widest spectrum of symptoms the ontological approach provides several methods to define a symptom: the simple threshold-based method, the complex expression-based method, and the method with external procedures.

The simple threshold-based method can be used, when a monitoring parameter has a fixed range of normal values and unambiguously gives evidence of a fault. A typical example would be a fan failure, which manifests in a zero fan speed. This simple method of symptom definition utilizes data properties of ontology: core ontology schema includes data properties *is_normal_above*, *is_normal_below*, and *is_normal_when* to be used with subclasses from *Parameter*. If a symptom of a fault includes several parameters, values of all parameters have to be abnormal to evidence a fault.

This threshold-based method has several obvious limitations. A data property can define normal values of a parameter only with a simple threshold or as a single value, but cannot be used in a straightforward manner if a range of normal values is a bounded interval or a tuple of values. Moreover, if one parameter has different ranges of normal values for different symptoms, has different significance for different symptoms or a fault manifests in different combinations of parameters' values, this method also fails.

The complex expression-based method of symptom definition overcomes all these limitations. An annotation property *has_symptom_description* for a subclass from *ComponentFault* can store an arbitrary expression to be interpreted in addition to *shows_in_parameter* object properties. In general, such an expression can take any form as long as it unambiguously describes a symptom. *shows_in_parameter* object properties although redundant for the fault identification task can still be used to identify, which parameters should be included in a set of monitored data.

For example, a disk is a component type, whose faults require this expression-based method of symptom definition. Disks perform complex self-diagnosing and report results that demand further analysis. Some of the complex symptoms (based on a report on failure trends by Pinheiro et al. [37]) are shown in Table 3.

When a symptom function is too complex to be expressed in an annotation property, a knowledge base can provide a link to an external procedure. This method with external procedures can also be used, when a fault manifests in some anomaly in the monitoring data and cannot be associated with immediate values of diagnostic parameters, which requires the application of machine learning techniques. In this case, a black-box diagnostic model would be able to detect changes in a component or subsystem performance, while corresponding segments of the ontology would be able to provide further insight into the problem.

Table 3

Examples of symptom descriptions in the ontology of data storage system health

Fault	Symptom description

Damage due to temperature changes	`<owl:Class rdf:about="urn:`**`AbnormalTemperatureDriveDamage`**`">`
	`  <has_symptom_description>` **`Grade`** `= consumer AND (`
	`  ` **`Temperature`** `< 30 OR` **`Temperature`** `> 40 AND` **`PowerOnHours`** `>= 3 y )`
	`  </has_symptom_description>`
	`</owl:Class>`

Damaged disc surface	`<owl:Class rdf:about="urn:`**`DriveSurfaceDefects`**`">`
	`  <has_symptom_description>` **`Grade`** `= consumer AND (`
	`  ` **`ScanErrorCount`** `> 0 AND` **`PowerOnHours`** `>= 1 y )`
	`  </has_symptom_description>`
	`</owl:Class>`

	`<owl:Class rdf:about="urn:`**`CriticalDriveSurfaceDefects`**`">`
	`  <has_symptom_description>` **`Grade`** `= consumer AND`
	`  ( ` **`ScanErrorCount`** `> 0 AND` **`PowerOnHours`** `< 1 m )`
	`  </has_symptom_description>`
	`</owl:Class>`

For these purposes additional *solves_with* and *described_by* pair of object properties is introduced as a method of formal description of implicit relations between diagnostic data and system health (see Fig. 4).

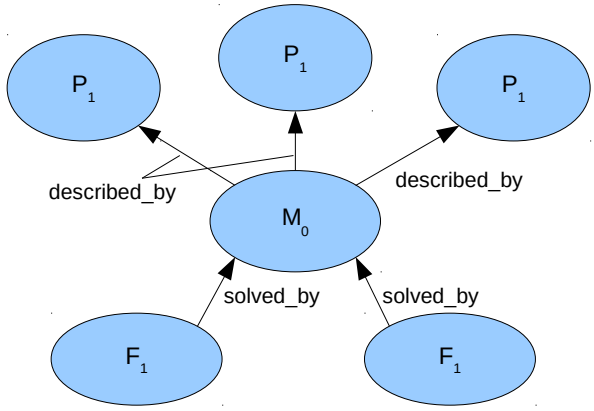

Figure 4
Example of symptoms description with links to external procedure

Properties *described_by* define a set of parameters *{P}* that should be passed to an external procedure M_0. An external procedure should classify the set of the parameter values as one of the probable component fault symptoms *{F}*, defined by *solves_with* property. In the simplest case, such a procedure contains a sequence of "if…else" rules. However, its main purpose is to provide an interface to the trained in advance machine learning models, such as anomaly detection or text classification methods. The assessment of fault event occurrence, made by the external procedure, is considered precise and unambiguous, although machine learning methods usually provide probabilities of various outcomes. In that case, component health is determined by the most probable fault event.

4 Software Logs

The presented paper proposes an approach to software log analysis, which can be used as an external procedure in the process of symptom definition. The proposed approach is based on text preprocessing and classification methods, widely used in the field of natural language processing. A pair of *solves_with* and *described_by* properties takes a set of software logs as diagnostic parameters *{P}*. The parameter values are determined using preprocessing of consecutive message blocks in a given time interval (one minute time interval prior to the current timestamp is recommended). The resulting set *{F, F'}* contains all probable fault events,

corresponding to the parameters from *{P} (F)* and an abstract event *(F')*, for a component state without any faults.

In such a manner a software log can be used as a diagnostic parameter only if it meets certain requirements: it should be stored in plain text and all log messages should have timestamps.

4.1 Log Data Preparation

The proposed fault detection algorithm contains three major steps:

1) Text preprocessing.
2) Calculation of the parameter values.
3) Classification procedure.

The text preprocessing procedure consists of the message text part extraction and text cleaning. Text cleaning is a procedure, commonly used in natural language processing. It includes text tokenization, non-numerical tokens removal, converting tokens to lowercase, removal of the stop-words, stemming, and filtration.

In the second step, the algorithm takes a resulting token array and calculates a final feature vector for each log, combining word embedding and the set of text-level features, presented in Table 4. Word embedding is a one-hot encoding with tf-idf [38] weight coefficients calculation. The text-level features characterize the statistical properties of the whole message block.

Table 4
Text-level log parameters

Parameter	Type
Number of tokens in a given time interval	Int
Number of messages in a given time interval	Int
Average log message length in a given time interval	Float
Average tokens per second in a given time interval	Int
Average messages per second in a given time interval	Int

Finally, the classification procedure uses a classifier pretrained to determine the most probable text class that corresponds to one of the probable fault events. Training and validation data sets should be prepared from log packages, collected during previous fault event occurrences.

4.2 Ontology Extension

The only step of the diagnostic procedure that requires some explicit log structure analysis is the process of splitting a log into a header and raw text parts in order to extract additional information such as timestamps, process identifiers, etc. Log message splitting is based on combinatorial parsing: each log header field has a simple parser, and those individual parsers should be applied in a defined consecutive order. The ontological model has been supplemented with a set of corresponding classes and properties to store the necessary information.

The Message header class *MessageHeaderFormat* contains the *MessageHeaderFormatField* subclass for information about header fields and their extraction priority and the *MessageHeaderFieldDataType* subclass with information about low-level field data types. Each low-level data type is linked to a dedicated individual parser that should be used to extract a header field value. See Fig. 5 with the example for the message.log file.

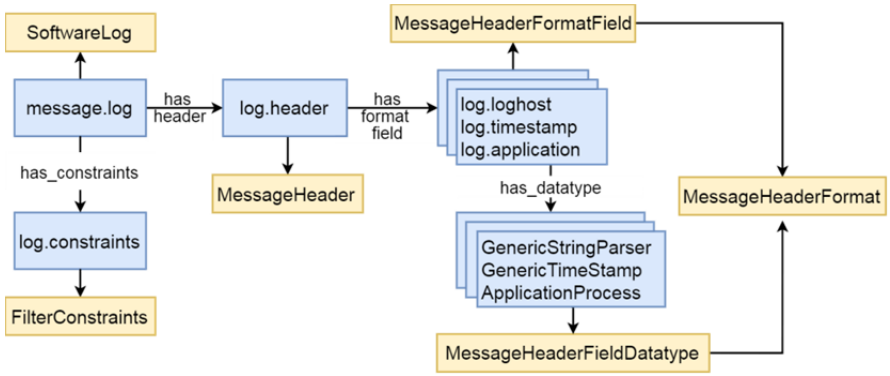

Figure 5

Segment of the ontology of data storage system health regarding message.log header extraction

4.3 Classification Results

The developed approach was evaluated on a dataset of ~ 6000 log packages, each containing up to 116 logs of 33 distinct types. This dataset contains logs, stored after the root cause of a fault investigation, mixed with logs for the healthy system. The dataset consists of 1600 “faulty” log packages with 41 different fault events caused by various problems in a data storage system configuration with two storage processors in a cluster and 16 disk enclosures.

The classification results, presented in Table 5, allow the conclusion that the most suitable classifier algorithm for the dataset is the Random Forest classifier.

Table 5
Classification results

Classifier	Average precision	Average recall	Average f-score	Model training time
Random Forest	0,74	0,71	0,71	183 s
Naïve Bayes	0,48	0,27	0,28	205 s
KNN	0,38	0,39	0,37	166 s
Logistic regression	0,28	0,33	0,28	498 s
SVM	0,23	0,26	0,20	430 s
LSTM	0,47	0,44	0,45	~4 h
GRU	0,24	0,22	0,23	~7 h
LSTM with attention	0,42	0,39	0,40	~ 6 h

Compared with the results acquired by Bertero [35], the presented approach is less accurate, but has the advantage of being able to process logs in almost all formats, analyze text instead of separate messages, and can be a part of an active diagnostic procedure.

Conclusions

The presented ontological approach to embedded diagnostics has been developed with the goal to support autonomous monitoring, diagnosis and self-management within a storage processor with the intent to complement the current data storage system diagnostic portfolio. This approach provides machine representation of the knowledge base in the domain of data storage reliability and diagnostics. The core ontology of the knowledge base provides a schema to store unstructured expert knowledge regarding typical redundancy schemes, symptoms of faults, possible causes of faults, their severity levels, and fault containment procedures. The resulting diagnostic model enables highly heterogeneous monitoring data as diagnostic parameters: for a data storage system they range in types of information, and their sources range in levels of the data storage system hierarchy.

A knowledge base converted from the ontology format to the graph database format is designed to be stored on a storage processor and updated by an administrator when necessary. An embedded implementation of diagnostic model provides necessary visibility of the internal organization of a data storage system. The graph database format provides an increased speed of queries processing. As a result, such direct access of the storage processor to the diagnostic model enables autonomous self-diagnostics at the rate of monitoring data updates. Another significant advantage of the proposed ontological approach is its flexibility, which allows constructing diagnostic models for arbitrary data storage systems.

The limitation of the presented approach is due to no inheritance from existing ontologies, which means that an editing expert has to make a preliminary acquaintance with the new schema of classes and relations.

Another limitation is that the utilization of ontology formalism does not fully comply with the standard OWL 2 specification. In particular, object properties are heavily used to specify relationships between classes, and domain and range properties are used to specify the range of definition for object properties. However, such non-standard use of ontologies is fully supported by the Protégé as an ontology editing tool and serves the objectives of the diagnostics. In this manner, an expert gets a convenient interface for knowledge base editing and after the ontology is converted to a graph database the ontological framework is not used for further processing.

Software logs have proven to be a valuable source of diagnostic data. The proposed approach to automated log mining performs on par with the most popular log analysis techniques without almost any specific research, which makes it an appropriate choice for an external procedure as a part of ontological fault symptom definition.

Thus, the new ontological approach to diagnose data storage systems is based on the formal model-based representation of expert knowledge about data storage system health and employs machine learning algorithms to define relations between health states and parameter values. Compared with the methods that implement only deterministic rules the new approach allows to automatically detect a wider range of fault types, whilst similarly to existing model-based approaches it allows to perform fault localization. The developed prototype of a diagnostic service allows for detection of more than 40 different fault types using log files as a source of diagnostic data, with an average precision of 0.74 and model training time of fewer than 4 minutes. The deterministic part of the implemented ontology-based diagnostic model contains around 2300 axioms, including 25 data and object property types and over 500 classes, with more than 100 referring to the faults in a data storage system.

Acknowledgement

This work was supported by the Ministry of Science and Higher Education of Russian Federation within the framework of the Federal Program “Research and Development in Priority Areas for the Development of the Russian Science and Technology Complex for 2014-2020” (RFMEFI58117X0023).

References

[1] U. Troppens, W. Muller-Friedt, R. Wolafka, R. Erkens, N. Haustein: Business continuity, Storage Networks Explained, 2nd ed., Chichester, United Kingdom, 2009

[2] Common architecture. Storage Management Initiative Specification (SMI-S) v1.8.0 rev. 3, Storage Network Industry Association, 2018

[3] J. Tate, P. Beck, H. H. Ibarra, S. Kumaravel, L. Miklas: Introduction to Storage Area Networks, IBM Redbooks, 2018

[4] O. Mamoutova, M. Uspenskiy, A. Sochnev, S. Smirnov, M. Bolsunovskaya: Knowledge based diagnostic approach for enterprise storage systems, Proceedings of IEEE 17th International Symposium on Intelligent Systems and Informatics, Subotica, Serbia, 2019

[5] V. Filks, G. Ruth: Magic quadrant for storage resource management and SAN management software, G00232275, Gartner, 2012

[6] G. Pruett et al.: BladeCenter systems management software, IBM Journal of Research and Development, Vol. 49, No. 6, 2005, pp. 963-975

[7] S. Gopisetty et al.: Evolution of storage management: transforming raw data into information, IBM Journal of Research and Development, Vol. 52, No. 4.5, 2008, pp. 341-352

[8] C. Ciccotelli et al.: NIRVANA: a non-intrusive black-box monitoring framework for rack-level fault detection, Proceedings of 21st Pacific Rim International Symposium on Dependable Computing, Zhangjiajie, China, 2015, pp. 11-20

[9] R. Kedia, A. Lunawat: Artificial intelligence based storage management architecture, Proceedings of 2018 IEEE International Conference on Cloud Computing in Emerging Markets, Bangalore, India, 2018, pp. 110-114

[10] 2019 top network performance challenges, Industry Survey Findings, Sirkin Research, 2018

[11] D. Breitgand, E. Henis, E. Ladan-Mozes, O. Shehory, E. Yerushalmi: Root-cause analysis of SAN performance problems: an I/O path affine search approach, Proceedings of 9th IFIP/IEEE International Symposium on Integrated Network Management, Nice, France, 2005, pp. 251-264

[12] R. Kannan, S. Ramanathan, S. Subramanian, B. Vaidhinathan: Root-cause approach to problem diagnosis in data networks, U.S. Patent 2009028053 (A1), Jan. 29, 2009

[13] N. F. Noy, D. Mcguinness: Ontology development 101: a guide to creating your first ontology, Stanford Knowledge Systems Laboratory, 2001

[14] M. Albert, T. Langle, H. Worn, M. Capobianco, A. Brighenti: Multi-agent systems for industrial diagnostics, IFAC Proceedings Volumes, Vol. 36, No. 5, 2003, pp. 459-464

[15] R. Muller: Ontologies in automation, M.S. thesis, TU Wien, Vienna, 2008

[16] M. Merdan, M. Vallee, W. Lepuschitz, A. Zoitl: Monitoring and diagnostics of industrial systems using automation agents, International Journal of Production Research, Vol. 49, No. 5, 2011, pp. 1497-1509

[17] L. B. Amor, I. Lahyani, M. Jmaiel: Towards ODRAH: an ontology-based data reliability assessment in mobile health, Proceedings of IEEE/ACS 13th

International Conference of Computer Systems and Applications, Agadir, Morocco, 2016, pp. 1-8

[18] M. Sir, Z. Bradac, V. Kaczmarczyk: Ontology and automation technique, Proceedings of 5th WSEAS international conference on Communications and information technology, Corfu Island, Greece, 2011, pp. 171-174

[19] F. Xu, X. Liu, W. Chen, C. Zhou, B. Cao: Ontology-based method for fault diagnosis of loaders, Sensors, Vol. 18, No. 3, 2018, p. 729

[20] D. G. Rajpathak: An ontology based text mining system for knowledge discovery from the diagnosis data in the automotive domain, Computers in Industry, Vol. 64, No. 5, 2013, pp. 565-580

[21] J. Yi, B. Ji, C. Chen, X. Tian: Employing ontology to build the engine fault diagnosis expert system, Proceedings of WRI World Congress on Computer Science and Information Engineering, Vol. 2, Los Angeles, USA, 2009, pp. 631-635

[22] M. T. Khadir, S. Klai: A steam turbine diagnostic maintenance system based on an evolutive domain ontology, Proceedings of International Conference on Machine and Web Intelligence, Algiers, Algeria, 2010, pp. 360-367

[23] A. Mallak, A. Behravan, C. Weber, M. Fathi, R. Obermaisser: A graph-based sensor fault detection and diagnosis for demand-controlled ventilation systems extracted from a semantic ontology, Proceedings of IEEE 22nd International Conference on Intelligent Engineering Systems, Las Palmas de Gran Canaria, Spain, 2018, pp. 000377-000382

[24] N. T. M. Saeed et al.: ADISTES ontology for active diagnosis of sensors and actuators in distributed embedded systems, Proceedings of IEEE International Conference on Electro Information Technology, Brookings, USA, 2019, pp. 572-577

[25] J. Schoenfisch, C. Meilicke, J. von Stulpnagel, J. Ortmann, H. Stuckenschmidt: Root cause analysis in IT infrastructures using ontologies and abduction in Markov Logic Networks, Information Systems, Vol. 74, May 2018, pp. 103-116

[26] N. Dendani, Med. Khadir, S. Guessoum: Hybrid approach for fault diagnosis based on CBR and ontology: Using jCOLIBRI framework, Proceedings of IEEE International Conference on Complex Systems, Agadir, Morocco, 2012, pp. 1-8

[27] S. H. El-Sappagh, M. Elmogy: Case based reasoning: case representation methodologies, International Journal of Advanced Computer Science and Applications, Vol. 6, No. 11, 2015

[28] What are linux logs? Code examples, tutorials & more, 2017, https://stackify.com/linux-logs/

[29] Marcel: 12 critical linux log files you must be monitoring, 2018, https://www.eurovps.com/blog/important-linux-log-files-you-must-be-monitoring/

[30] Linux logging basics – the ultimate guide to logging, https://www.loggly.com/ultimate-guide/linux-logging-basics/

[31] J. Zhu et al.: Tools and benchmarks for automated log parsing, Proceedings of IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, Montreal, Canada, 2019, pp. 121-130

[32] A. Pande, V. Ahuja: WEAC: word embeddings for anomaly classification from event logs, Proceedings of IEEE International Conference on Big Data, Boston, USA, 2017, pp. 1095-1100

[33] P. He, J. Zhu, S. He, J. Li, M. R. Lyu: An evaluation study on log parsing and its use in log mining, Proceedings of 46th Annual International Conference on Dependable Systems and Networks, 2016, pp. 654-661

[34] H. Hamooni et al.: LogMine: fast pattern recognition for log analytics, Proceedings of 25th ACM International Conference on Information and Knowledge Management, New York, USA, 2016, pp. 1573-1582

[35] C. Bertero: Experience report: log mining using natural language processing and application to anomaly detection, Proceedings of IEEE 28th International Symposium on Software Reliability Engineering, Toulouse, 2017, pp. 351-360

[36] M. A. Musen: The Protege project: a look back and a look forward, AI Matters, Vol. 1, No. 4, 2015, pp. 4-12

[37] E. Pinheiro, W.-D. Weber, L. A. Barroso: Failure trends in a large disk drive population, Proceedings of 5th USENIX Conference on File and Storage Technologies, Berkeley, USA, 2007, p. 2

[38] G. Salton, C. Buckley: Term-weighting approaches in automatic text retrieval, Information Processing & Management, No. 24, 1988, pp. 513-523