Serdica J. Computing **4** (2010), 243–262

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

SEMANTICALLY ENHANCED SOFTWARE DOCUMENTATION PROCESSES

Werner Klieber, Michael Granitzer, Mansuet Gaisbauer, Klaus Tochtermann

Abstract. High-quality software documentation is a substantial issue for understanding software systems. Shorter time-to-market software cycles increase the importance of automatism for keeping the documentation up to date. In this paper, we describe the automatic support of the software documentation process using semantic technologies. We introduce a software documentation ontology as an underlying knowledge base. The defined ontology is populated automatically by analysing source code, software documentation and code execution. Through selected results we demonstrate that the use of such semantic systems can support software documentation processes efficiently.

1. Introduction. Developing large software systems remains a complex, error prone task. High quality documentation is essential for understanding the intention of a software system. As outlined in [8], good documentation provides multiple complementary views on the software system. The documentation has

ACM Computing Classification System (1998): D.2.7, K.6.3, H.2.1, H.2.8, F.2.2.
Key words: Documentation, Semantics, Software Engineering.

to reflect the developer's view on how the system works and provides logical views on the system for users as to how to consume the provided functionality.

However, the establishment of new software processes requires changes in the corresponding documentation processes. Agile software development methods such as SCRUM [17] develop software in "sprint" periods taking typically one to four weeks. Such a short, dynamic software development process requires more agile and flexible software documentation to efficiently support short development periods. Furthermore, software documentation has to be coherent and semantically accurate [7]. Hence, automatic support in software documentation increases accuracy and reduces workload on developers.

Another change observed in developing large software systems is the use of service-oriented architectures: loosely coupled, decentralized components with commonly agreed contracts and data exchange formats work together orchestrated by a workflow engine. As experience with large service-oriented architectures has shown, complexity increases with the number of loosely coupled components [15]. While ontologies allow semantically rich definitions of contracts between services, thereby increasing the automatic support in workflow orchestration, modelling such ontologies is labour-intensive. Detailed documentation of components and a deep understanding of the correct service usage as envisioned by the developers support the construction and maintenance of large, decentralized ontology-driven service oriented architectures.

In our work we propose an ontology-based system for agile, collaborative and accurate software documentation. We develop an ontology to gather static information like software source code, module dependencies, collaborative information like bug reports, as well as runtime information mined via analysing code/service executions. We focus on automatic tools to populate the ontology from different sources including runtime execution logs. The populated ontology is further analysed via process-mining methods to detect meaningful, potentially useful process patterns [16]. Through the clear defined semantic, the populated ontology can be reused for efficient collaborative software documentation using Semantic Wikis [4] or for automatic processing of software documentation. Through making machine-readable data about software publicly available, new means for service usage analysis and service orchestration could be developed.

The paper is structured as follows. Chapter 2 motivates the use of semantic technologies for software engineering processes and introduces the conceptual architecture of our system. Chapter 3 introduces the software ontology, the ontology modelling process we used, and discusses the design in detail. In Chapter 4 we describe the process and technical impacts of populating a semantic repository. In Chapter 5 we describe some scenarios to outline the practicability of ontologies for an automated documentation process.

2. Ontology-based Software Documentation. Software documentation is closely related to knowledge management. It requires collecting knowledge from various, usually heterogeneous, semi-structured sources and representing it in a formalized way. Software development involves a large set of different tools used by development teams: Version control systems manage source code versions; Continuous integration systems test and deploy software modules frequently; Repository managers maintain single software modules for downloading; Code quality management tools analyse source code against code metrics to ensure reliable and robust software; Source code documentation tools provide technical descriptions about the software created form the source code.

Collecting information from all of these heterogeneous systems in a machine-processable form is cumbersome and usually involves data mining techniques for resolving conflicts. Providing all information in a technology-independent and self-describing repository is mandatory for all-encompassing software development processes.

A standardized and flexible data environment for the gathered information enables unambiguous usage for every tool involved in the documentation process. Ontologies are a well investigated, accepted approach for formalizing shared concepts making them understandable by both humans and computers. Figure 1 shows a high-level architecture of our framework used for analysing source code. We use a central semantic repository described by an ontology to harmonize all needed information.

This central repository facilitates accessing the information in a tool-independent way. Independently developed tools can be used to gather the needed information from the various data sources and store them consistently in the semantic repository.

Report-generating or analysing tools can consume this data and store their results back in the repository if needed. Semantic repositories allow the same activities like storing, querying and managing of structured data as other data management systems. The major benefits of semantic repositories can be summarized as follows [5]:

- Semantic repositories use generic and flexible data models like graphs. This facilitates interpreting and adopting the stored information easily.
- Stored data is described according to semantic schemata. This allows automatic reasoning about the data and harmonizing data from heterogeneous data sources.

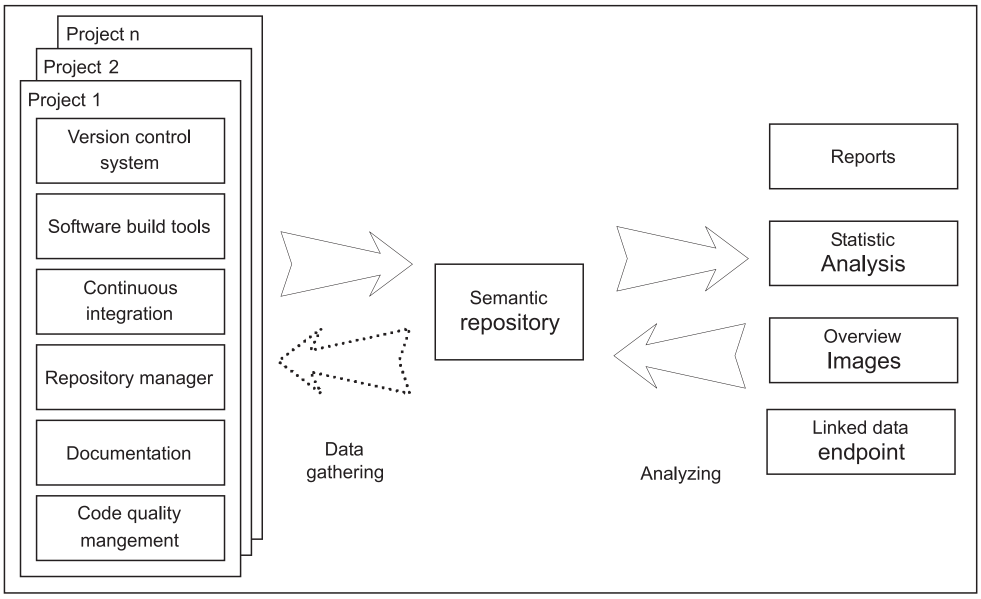


Fig. 1. Conceptual architecture using a central semantic repository

3. The software ontology. The expressiveness of the ontology used in the central software repository directly influences the applicability of the system. In this section we introduce the main goals and aspects of our software ontology.

Details about the modelling process can be found in [10]. To keep the vocabularies and concepts of the ontology consistent, the ontology has been developed from scratch instead of reusing existing ontologies. Single parts are designed by reusing design concepts from existing tools (e.g. project dependency tools). To exploit synergies with already existing ontologies we follow the linked open data principle by using an "equals-to" relation to map among equal concepts in different ontology schemas. In a final step, an extra domain-specific ontology has been derived from the core ontology filled with domain-specific instances applying to our testing scenario. For instance, all employees working in our company have been covered. These persons are used to focus on people of interest for reports.

The ontology can be grouped coarsely into three parts: project metadata, static source code and dynamic usage traces. Figure 2 shows the main concepts of the ontology. The concepts are motivated by recurring documentation needs from developers. These needs are:

- Software project specific metadata is modelled to gain descriptive information about a project itself and its dependency to other projects.

- Static source code information is modelled to mine data on a source code level into a semantic repository.
- Dynamic program execution events are modelled to mine data during program execution.

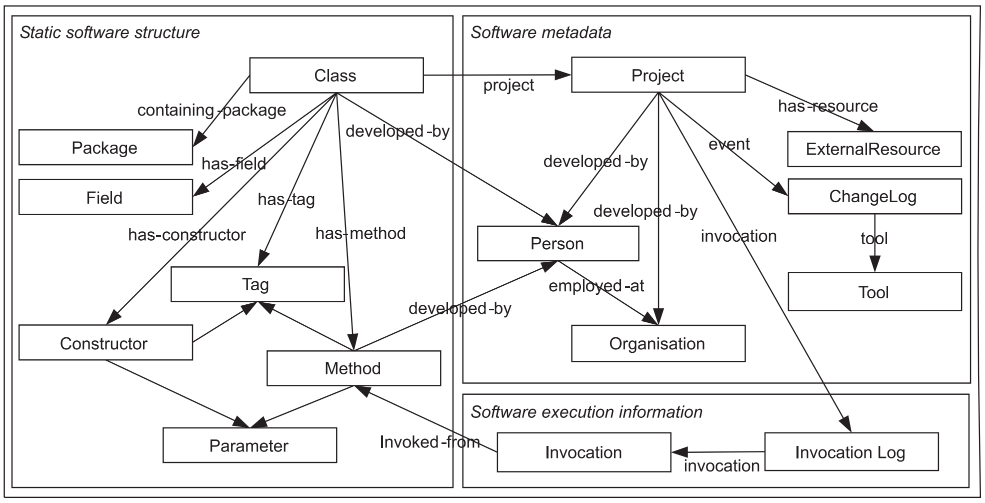


Fig. 2. Overview figure of the software ontology showing the most relevant concepts and properties[1]

3.1. Project metadata. The project concept allows expressing project metadata in the ontology and groups all project-related aspects together. This project-related part contains (i) dependencies among projects using parent-child relationships, (ii) concepts on used software development and reporting tools like revision control systems, bug tracking systems, automated build and release management systems, reporting tools, etc., (iii) dependencies to social structures like persons or organizations, as well as (iv) dependencies to external resources like images. Modifications in the repository can be archived in a change Log. Querying this change log archive allows tools to validate that all required information has already been added to the repository in preliminary stages.

3.2. Static source code. Analysing static source code is an important information source for understanding the architecture and concepts behind a software system. Selective classes, operations and source code comments can be used

[1]The ontology is available for download from our web site:
`http://www.know-center.at/ontologies/2009/2/software-project.owl`

to represent the current state of an API. The ontology models the source code structure reflecting characteristics of object oriented programming languages. In particular, ontology models the relationship of classes, methods, variables and package structure of the source code. All of these source code elements can be tagged with keywords. The concept of "tagging" resources can be used by developers to add extra, non-program information directly into the source code. The ontology distinguishes between productive source code and test code. The productive source code contains all source code dealing with the program execution logic. The test code contains all tests written by developers. Beside using test code as an indicator for test coverage of program functionality, selective tests are good usage examples for application developers. Using the test code itself as usage examples ensures that the source code examples are up to date and maintained. Inspecting test code allows more in-depth knowledge about a software system.

3.3. Dynamic Program execution. The ontology provides concepts to store software execution events in a semantic repository logged during program execution. Analysing this information allows more in-depth knowledge on the order in which the operations need to be executed to reach certain goals.

- Analysing the operation execution time is a useful indicator for performance.
- Analysing operation errors is a useful stability indicator.
- Analysing operation usage count: that an operation is called more frequently might be an indicator about its importance.
- Analysing the amount of times an operation is called within tests is a simple indicator for test coverage.

Approaches in the business process mining research can be used for analysing these events for meaningful patterns and workflows. The concepts modelled in the ontology are based on the data format used by the business process mining tool ProM[2]. Occurring events are modelled as a sequence list. Each sequence list is assigned to a unique usage case. Three types of events can be distinguished in the context of tracing software operation calls:

- Begin: this event occurs when initiating an operation call.
- End: this event indicates finishing an operation call.

[2]The ProM Framework: `http://prom.win.tue.nl/tools/prom/`

- Error: this event indicates operation calls causing exceptions.

Each invocation event contains a timestamp of its occurrence, the memory consumption, the originator of the event and a process instance name.

4. Populating the software ontology. Populating the software ontology from various data sources into one common format involves two main aspects. The first aspect is – given the software ontology – mainly an engineering task accessing required data from various source repositories and converting the data into a new format. For instance, reading source code files from a version control system and converting them into a set of RDF concepts of classes, methods, parameters and comments. A second aspect is extracting additional information from data not explicitly available. For instance, identifying the author of a source code class and assigning the author non-ambiguously to persons in the target ontology. Analysing runtime execution information to gain software usage information may involve statistical analysing algorithms.

4.1. Mining static software information. An important task when mining static source code is disambiguating single information artefacts. Various tools exist to help parsing the structured content within source code files proper. In the Java programming language for example structured documentation is managed via the Javadoc tool. However, gathering additional information such as the author of a file and the date is a non-trivial task.

Most integrated development environments use templates to automatically create structured documentation (including author's name, date, etc.) upon creating new classes. However each tool has its own conventions for naming and storing this information. For example, development environments usually use the login name as author identification. When developers keep the default settings, their name needs to be mapped to real persons' names before being useful.

The date format mostly depends on the local settings of the system, e.g., day-month-year in German-speaking countries and month-day-year in the USA. Furthermore it is likely that some source code contains no such information or that the information is incorrect, as when the developer has copied and pasted the information from another location and forgotten to update the information.

Similarly, author and date information can be obtained from source code versioning systems. Using the person and date information when the file has been first added into the version control system could serve as a source of evidence for the author and date information.

However, this information may be incorrect. The person who originally checked in a file in the version control system may not be author but a system designer setting up the project or the administrator moving projects between version control systems. Hence, harmonizing the different evidence sources is necessary.

When parsing our source code repository we used a manually created lookup table to map the author information given in source code headers to the persons in our semantic repository. In our evaluation only a few mappings could not be resolved automatically. Two developers with the same first name have used their first name in some header files. Since they worked on different projects and based on version control information these ambiguities have been resolved by directly replacing their name in the source code header.

We used statistical methods to analyse the gathered information to resolve dependencies among correlating information. Co-occurrence matrices were calculated to discover associations between terms. We used unsupervised machine learning algorithms to cluster source code artefacts using the k-means algorithm.

In the technical realization, first the RDF repository was queried for source code classes containing author and comment information resulting in 2686 classes. We used package names, class names and method names as additional information. This information increased the quality of the results significantly since developers organize and name their code according to the task each piece has to perform. An author-term co-occurrence matrix was built using the authors of a class as one dimension and the terms as the other. Next the classes were clustered and the top terms of each cluster were used to search for most matching authors in the co-occurrence matrix. Finally, a stop word file was filled manually to improve the results. Technical terms such as “enum” or “iterator” and some html-tag relicts like “<code>null</code>” were filtered out.

4.2. Mining dynamic runtime execution information. Collecting runtime execution sequences provides meaningful information about the usage of a system. To collect the needed execution data three intervention levels with the observed system can be distinguished.

- Source code level: At distinct points in the source code logging instructions can be added to gather the needed information.
- Compile time level: At distinct points in the compiled code a logging instruction is injected.
- Execution level: Program execution is interrupted at distinct source code points to gather the needed information. Here the same mechanisms as used by debuggers are applicable.

Collecting information at a source code level prohibits mining for third party modules with non-accessible source code. Therefore we did not investigate this approach. Collecting the information at execution level was our first choice because we can use the same mechanisms as debuggers do. Therefore the needed functionality should be well integrated in programming platforms. In our case we were interested in collection information from Java programs.

The Java development platform provides a Java Platform Debugger Architecture (JPDA). This framework provides a high level debugging API named JDI. This API was used to fetch "before method" and "after method" invocation events using TCP[3] transport mode to connect. However, in this setting the execution of the traced program slowed down significantly. For instance, logging 60 events increases the execution time from 4 seconds to 10 minutes. Several reasons for this slowdown can be detected. Using the debugging interface disables the Java just-in-time compiler. The high level API fetches all messages and filters them out later. The framework also provides a low level native interface. However, implementing this API is time-consuming. Therefore we collected the invocation events at compile time. We used the aspect-oriented programming tool AspectJ [1] to weave required logging aspects into compiled Java classes. For performance reasons the data is written in an internal binary format to the file system. In an extra step the binary data is converted to its RDF representation and imported into the semantic repository.

We focus on collecting sequences covering two aspects:

- *High level Workflow Analysis*: Collect execution events from high-level interfaces frequently used by application developers. Analysing sequences from high-level interfaces – like for example web service calls – provides useful information on how to use an interface or a set of services efficiently and how to identify best practice workflows.

- *Module State Transition Analysis:* Collect execution events from the modules involved when executing operations from high level interfaces. Analysing low-level sequences provides useful information on how the system operates internally and which state transitions take place.

Details on the mining methods, application and usage scenarios are provided in the next sections.

[3]The JPDA architecture ships with two data transport implementations: a shared memory transport (for Windows only) and TCP/IP network transport implementation. For details see `http://Java.sun.com/Javase/6/docs/technotes/guides/jpda/conninv.html`

5. Application Scenarios & Evaluation of static software information. A common argument for using semantic technologies is its flexible and adaptive behaviour. The information stored in a data management repository is influenced mainly by the needs of the tools using it. For human users it is essential to provide multiple views on a complex software system for deeper understanding. So a semantic repository in our context needs to be flexible enough to store all software engineering data. Through such a flexible semantic repository data from all sources can be queried to automatically generate documentation artefacts to be included in various documentation processes. Supported through mining techniques as outlinedabove, patterns can be revealed which are not directly visible to humans. In this section we describe different application scenarios which have been implemented.

5.1. Sample Data. The applicability of the ontology has been tested by analysing the source code of our knowledge discovery framework "KnowMiner" [11].

This project was used because of its size and our knowledge on the persons and processes involved in its creation. Furthermore, the results can easily be interpreted according to its correctness. The KnowMiner framework is a service-oriented architecture enabling different teams – all experts in their research areas – to work on isolated components. Whereas special effort has been undertaken in providing a simple-to-use API to application developers, an open issue for new developers is how to assemble their workflows.

The KnowMiner framework uses the Maven project management tool from the Apache software foundation to define all of its dependent modules and third party libraries. These Maven project files are parsed, converted to RDF data in the ontology schema and imported to the triple store. The conversion is done using an adapted version of the Java2rdf tool.[4] The KnowMiner Framework contains about 400.000 lines of code, separated in 74 modules and 32 test modules and 65 third party libraries. For our experiment we used the TDB Jena triple store.[5] Jena TDB is a file-based storage backend for the Jena semantic web backend. After populating the sample data the database contains about 4.5 million triples. The next sections introduce some usages scenarios for finding and presenting relevant information.

5.2. Application Scenario Overview Charts. A first example illustrating the usage of the ontology is generating simple overview charts. When developers start working with the KnowMiner framework, they start with simple

[4]Java RDF documentation tool: `http://simile.mit.edu/wiki/Java_RDFizer`.

[5]TDB – A SPARQL database for Jena: `http://jena.sourceforge.net/TDB/`.

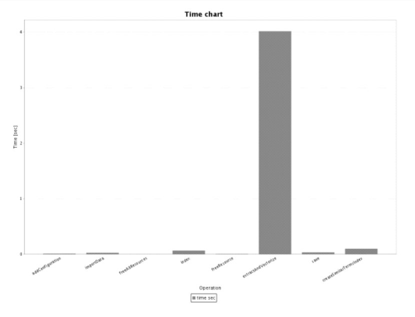
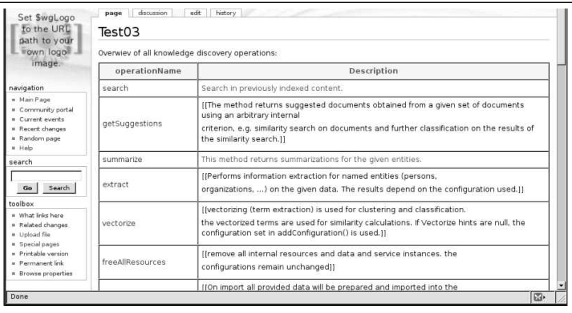


Figure a



Test03

Overwiev of all knowledge discovery operations:

operationName	Description
search	Search in previously indexed content.
getSuggestions	[[The method returns suggested documents obtained from a given set of documents using an arbitrary internal criterion, e.g. similarity search on documents and further classification on the results of the similarity search.]]
summarize	This method returns summarizations for the given entities.
extract	[[Performs information extraction for named entities (persons, organizations, ...) on the given data. The results depend on the configuration used.]]
vectorize	[[vectorizing (term extraction) is used for clustering and classification. the vectorized terms are used for similarity calculations. If Vectorize hints are null, the configuration set in addConfiguration() is used.]]
freeAllResources	[[remove all internal resources and data and service instances. the configurations remain unchanged]]
	[[On import all provided data will be prepared and imported into the

Figure b

Fig. 3. Figure 3a illustrates a sample diagram showing the average execution time of an operation in seconds. Figure 3b illustrates presenting RDF query results in a Wiki environment.

workflows, make them running and then add some more complex functionality. When adding further functionality a common question is the effect of the whole workflow on the execution time. Figure 3a shows an overview diagram of the average execution time of each method obtained through mining invocation logs. This diagram helps developers to estimate performance issues. Mostly they start with a simple information retrieval task: import some data and store it in an index to make the data searchable. Next some extraction tasks are included, for instance generating vectors so the data becomes comparable, as when estimating the similarity between documents. According to the diagram in Figure 3a this extraction task takes much more time than the other tasks. Sometimes application developers are insecure about the long execution time when using the extract operation the first time. A look at the diagram helps them to understand that this is a technical issue.

Such overview diagrams are furthermore useful for finding out which operations are good candidates for optimizing.

5.3. Application Scenario Web front-end. Wikis are frequently used to provide information on web pages that are easily producible and maintainable. Presenting some software project information gathered from various sources allows one single point of access for developers to get the needed information. Semantic wikis provide a useful extension to normal wikis by enriching content and wiki pages with semantic concepts. This provides a flexible way to provide overview pages collecting information according to specific aspects. Especially in the context of software documentation multiple views on different aspects of a complex software system are essential.

Using automatic generated content to show in a wiki simplifies the report generation and reusing the information. Changes will show up automatically in the wiki pages when the software project evolves.

Figure 3b shows a screenshot of the API from our knowledge discovery framework. It lists all operations of the main interface and the source code documentation provided by the developers within the source code. For the technical realization the semantic media wiki framework is used. An RDF plugin has been developed to send SPARQL queries embedded in a wiki page to the RDF backend.

Source code documentation tends to be very technical and hard to understand without knowing the local context in the source code. Showing the documentation in a web front-end where users can give feedback to developers about the understandability and accuracy of their documentation can help to improve the quality of the source code documentation.

5.4. Extraction of Application Scenario Developer Expertise. In an organizational context it is essential to detect the topics covered by a software system and how they agree with the business topics. Knowing which persons engage with these topics is meaningful information for making decisions.

For our experiment we used our sample data to find out what are the most relevant topics covered by the source code and which developers address these topics. The outcome our statistical analysing mining approach is visualized in a simple list showing the top ten terms of each cluster and the top two persons assigned to it. The stop word list was filled manually with 19 entries to improve the results. The result is shown in Figure 4.

The results contain some useful insights into the software system:

- The clusters are well separated according to the main topics that our department deals with.
- Usually the two persons assigned to each topic cluster work together on these topics. This is useful information when one person is no longer available and a topic needs to be reassigned.
- Two people on the list no longer work at the company and one person moved to a management position. So this overview gives useful hints to reassign unmaintained source code.
- People appearing more frequently have overview knowledge about the system. They are good candidates to ask overview questions. Sparsely appearing persons are the experts on single topics and can be asked detail questions.

Our experiment shows that detecting topic and assigning them to persons generates meaningful results. However, the topic labels are unsatisfactory when describing the identified topics. Here the same problem as mentioned in [4] occurs. It is hard to create a meaningful label from a topic automatically. In many cases the terms are abbreviations or acronyms of business topics. Manual labelling has to be done to represent domain topics adequately.

6. Application Scenarios & Evaluation of runtime software information. For our experiments we used the invocation logs generated by executing some unit tests from our knowledge discovery framework. The intention is to get useful workflows assembled from developers who know best how to consume the functionality. The first results showed that the complete invocation events from all unit tests contain too much noise to detect meaningful workflows. One reason is that many tests produce no usable workflows with regard of meaningful API usage. For example some tests call an operation several times with the same input data to ensure the same results are delivered each time. Such a usage represents an atypical workflow since users can expect that an operation will work correctly. Furthermore it is difficult to detect the start and end points of a workflow correctly.

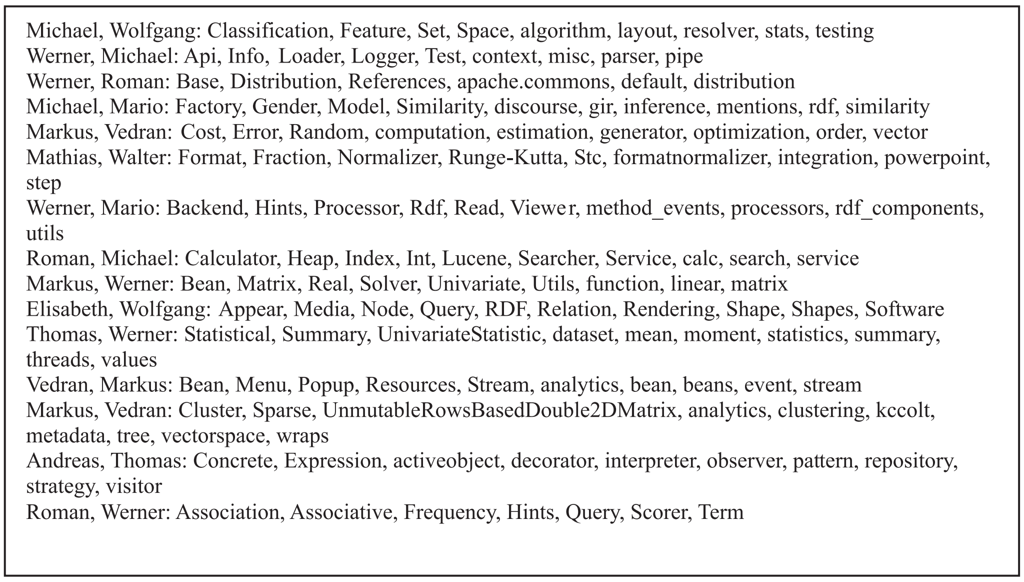

Michael, Wolfgang: Classification, Feature, Set, Space, algorithm, layout, resolver, stats, testing
Werner, Michael: Api, Info, Loader, Logger, Test, context, misc, parser, pipe
Werner, Roman: Base, Distribution, References, apache.commons, default, distribution
Michael, Mario: Factory, Gender, Model, Similarity, discourse, gir, inference, mentions, rdf, similarity
Markus, Vedran: Cost, Error, Random, computation, estimation, generator, optimization, order, vector
Mathias, Walter: Format, Fraction, Normalizer, Runge-Kutta, Stc, formatnormalizer, integration, powerpoint, step
Werner, Mario: Backend, Hints, Processor, Rdf, Read, Viewer, method_events, processors, rdf_components, utils
Roman, Michael: Calculator, Heap, Index, Int, Lucene, Searcher, Service, calc, search, service
Markus, Werner: Bean, Matrix, Real, Solver, Univariate, Utils, function, linear, matrix
Elisabeth, Wolfgang: Appear, Media, Node, Query, RDF, Relation, Rendering, Shape, Shapes, Software
Thomas, Werner: Statistical, Summary, UnivariateStatistic, dataset, mean, moment, statistics, summary, threads, values
Vedran, Markus: Bean, Menu, Popup, Resources, Stream, analytics, bean, beans, event, stream
Markus, Vedran: Cluster, Sparse, UnmutableRowsBasedDouble2DMatrix, analytics, clustering, kccolt, metadata, tree, vectorspace, wraps
Andreas, Thomas: Concrete, Expression, activeobject, decorator, interpreter, observer, pattern, repository, strategy, visitor
Roman, Werner: Association, Associative, Frequency, Hints, Query, Scorer, Term

Fig. 4. Result of a topic detection showing clustered source code keywords assigned to the most matching authors

Detecting these boundaries within one large sequence automatically is error prone. For instance, typically at the end of a workflow some resources are cleared and at the beginning of a workflow some data is loaded. Therefore it is likely that algorithms will detect the end of a workflow and the beginning of the next workflow falsely as a meaningful frequently used pattern. Using time information is not applicable in situations where workflows are triggered by automatic processes since there is no time shift between the end of one workflow and the start of the next workflow. Considering each test itself as one single workflow is error-phone. For performance reasons some developers reuse results among their tests. Using each single test in isolation will result in partly invalid workflows.

Tests are written either to test some functionality or to act as exemplary usage code. Only tests intended to demonstrate the usage are interesting for analysing workflows. Including all kind of tests worsens the results. For our experiments we used a human-assisted approach to select useful tests. Tests of interest were marked in the source code to be filterable by SPARQL queries.

6.1. Application Scenario Workflow Analysis. For our first experiments we used the ProM process mining tool. The ProM framework provides a large set of state-of-the-art process mining algorithms that can be used by an intuitive graphical user interface. It is designed to help managers to figure out how a process executed by employees harmonizes with predefined process models. In the context of software engineering we use this tool to analyse the process execution based on program execution events. The program execution events in the semantic repository are converted to the XML format (MXML) used by ProM tool.

Several iterations of filtering input data and trying multiple algorithms needs to be done to achieve useful results. Figure 5 shows a state diagram of our knowledge discovery framework generated with the alpha++ algorithm of the ProM framework [14]. The input data contains 130 events from 15 operations invoked of our main API recorded from 46 test executions.

The generated diagram provides an overview of our pre-processing workflow, illustrating the main pattern of its usage. For instance, first a configuration must be added to specify domain-specific settings so the framework can work. Next the “import data” operation can be used as data source to load some operational data into the system. After performing some algorithmic tasks at the end of a workflow the system needs to be cleaned up by freeing no longer needed resources.

ProM is a sophisticated tool to analyse business processes. It comes with

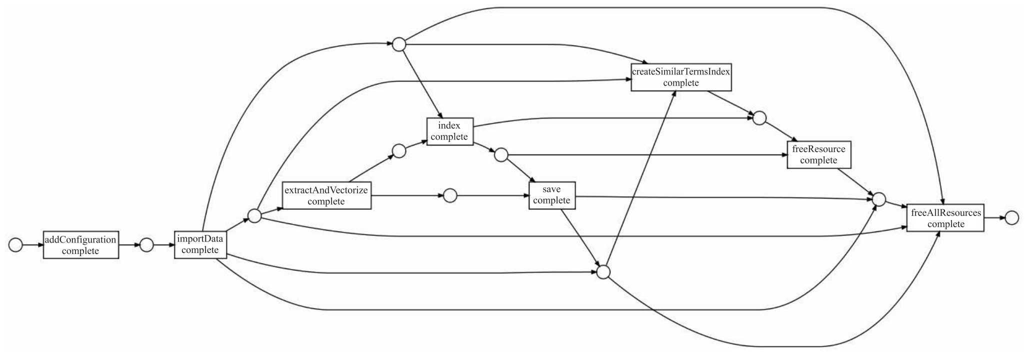

addConfiguration complete
importData complete
extractAndVectorize complete
index complete
save complete
createSimilarTermsIndex complete
freeResource complete
freeAllResources complete

Fig. 5. A sample state diagram created using the alpha++ algorithm from the ProM Framework

a large set of algorithms and analysing facilities. However, for more in-depth examination of software engineering behaviour specialized tools are needed.

6.2. Application Scenario Transition diagrams. Using transition diagrams is a robust approach to get an overview of valid sequence alternatives. An adjacent matrix is calculated by registering all direct succeeding events in the sequence log.

The transitions can be interpreted as useful assembling options. The transitions do not necessarily contain all valid connection possibilities between components when not covered by the input data. Figure 6 show a result image visualizing the invocation chronology of our information extraction pipeline.

The image is rendered using a force directed placement algorithms to group the components automatically. More frequently invoked components are drawn as visual larger nodes. Presentable labels have been calculated by applying filters to get rid of technical details, i.e., "at.knowcenter.ie.opennlp.PartOfSpeech Annotator.annotate()" is converted to "part of speech annotator". The calculated sequence is closed because for a given document the pipeline is executed multiple times for single document fragments. Rendered images are stored on a web server and linked as an external resource into the semantic repository.

Transition analysis is a useful approach to getting an overview of allowed states between single components. For more detailed information like a certain path to follow for solving a given problem, a more sophisticated analysis is needed. The discussed approach will deliver no meaningful results when the recorded sequences belong to different architectural layers. This means when single components are wrapped by parent components and the execution among these components changes depending on internal branch conditions. In a next

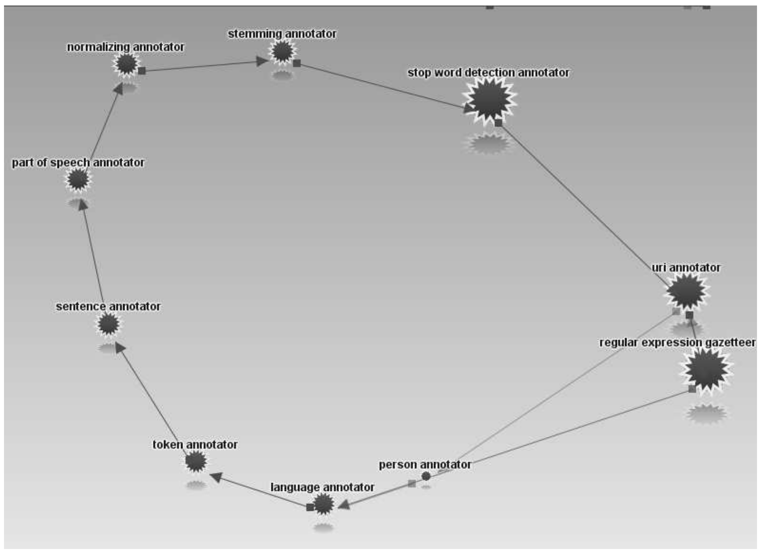


Fig. 6. A transition diagram showing the invocation sequence of a named entity extraction pipeline

step we will enhance our approach to support hierarchical execution traces, i.e., by using execution time. When the start and end time of an operation lies between the start and end time of another operation, the second operation invokes the first one during its execution. This enables building up a parent-child hierarchy.

7. Related Work. Recently in the software engineering area much investigation has been undertaken to mine and analyse software-related information to expose internal processes.

Software engineering deals with structured data like source code or runtime trace calls. It involves informal data like documentation, bug entries or mail communication. The data can be represented in graphs—i.e., for graph calls, sequences, or for execution traces of informal text [19]. For instance [2] annotate bug reports enabling developers to find interesting information artefacts more easily: patches, stack traces, source code and enumerations describing causalities. In [3] the email communication of the open source project Apache is analysed to span up a social network among all participants.

According to Conway's law there is a relationship between the technical structure and the organizational structure within an organization. Research projects have already shown that such a relationship can be computed [18]. The correlation between developers based on bug entries is compared with the maintainers of software components. Maskeri et al. [13] propose a human-assisted approach to extract topics from source code. The Latent Dirichlet Allocation model is used to group keywords in the source code to topics and label them manually.

Several investigations have shown the usefulness of ontologies to support software engineering processes. Cortellessa [6] introduces a software performance ontology to harmonize performance indicators used by different approaches. Ambrosio [1] developed an ontology-based documentation tool to archive documentation artefacts in a cohesive and historical comprehensible way.

When presenting various information types to humans, an intuitive and consistent medium is essential. Wikis came up recently as another effective collaboration and knowledge management tool due to their simple usage.

The survey of [12] states that companies use wikis frequently. One of the most common activities wikis are used is for software documentation processes.

The survey reports three types of benefits of wikis: enhanced reputation for users, making work easier and helping organizations improve their processes. However, wikis formalize knowledge on a rather low degree. This open structure creates the main problem of wikis: making navigation and search difficult [4]. This motivates enriching wikis with semantic technologies to overcome these limitations.

8. Conclusion. In this paper we presented a software ontology to model concepts occurring in the software documentation process. Using a unified, ontology-based data format for all tools involved in the automatic documentation process helps avoiding impreciseness in workflows and enables machine-understandable specification of data and operations. In our usage examples we demonstrated the feasibility of using an RDF triple store, ontologies and SPARQL queries to support an automatic documentation generation process. Using generic data formats like RDF triples provides a flexible data characteristic adjustment on software evolvements processes. Further, those standardized formats allow in-depth analysis of code relationships and their presentation in collaboration-enhancing tools like wikis.

However, SPARQL still misses some necessary features, which complicates its usage. For instance aggregator functions to generate sums on query results commonly needed for report generation are missing.

In a next step we plan to make our software framework open-source, and provide its documentation semantically enriched in the linked open data cloud.

Acknowledgements. The Know-Center is funded within the Austrian

COMET Programme – Competence Centres for Excellent Technologies – under the auspices of the Austrian Federal Ministry of Transport, Innovation and Technology, the Austrian Federal Ministry of Economy, Family and Youth and by the State of Styria. COMET is managed by the Austrian Research Promotion Agency FFG.

REFERENCES

[1] Ambrosio A., D. de Santos, F. de Lucena, J. da Silva. Software engineering documentation: an ontology-based approach. In: WebMedia and LA-Web, 2004, Proceedings, 38–40.
`http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1348143`

[2] Bettenburg N., R. Premraj, T. Zimmermann, S. Kim. Extracting structural information from bug reports. In: Proceedings of the 2008 international workshop on Mining software repositories – MSR'08, 27–30.
`http://portal.acm.org/citation.cfm?doid=1370750.1370757`

[3] Bird C., A. Gourley, P. Devanbu, M. Gertz, A. Swaminathan. Mining email social networks. In: Proceedings of the 2006 international workshop on Mining software repositories – MSR'06, 137–143.
`http://portal.acm.org/citation.cfm?doid=1137983.1138016`

[4] Buffa M., F. Gandon. SweetWiki: semantic web enabled technologies in Wiki. In: Proceedings of the 2006 international symposium on Wikis, ACM, 69–78. `http://portal.acm.org/citation.cfm?id=1149453.1149469.`

[5] Casanave C. Designing a Semantic Repository.
`www.w3.org/2007/06/eGov-dc/papers/SemanticRepository.pdf`, February 2010.

[6] Cortellessa V. How far are we from the definition of a common software performance ontology? In: Proceedings of the 5th international workshop on Software and performance – WOSP'05, 195–204.
`http://portal.acm.org/citation.cfm?doid=1071021.1071044`

[7] Hartmann J, S. Huang, S. Tilley. Documenting software systems with views II: an integrated approach based on XML. In: Proceedings of the 19th

annual international conference on Computer documentation, ACM, 2001, 237–246. `http://portal.acm.org/citation.cfm?id=501571`

[8] Huang S., S. Tilley. Towards a documentation maturity model. In: Proceedings of the 21st annual international conference on Documentation – SIGDOC'03, 93–99.
`http://portal.acm.org/citation.cfm?doid=944868.944888`

[9] Kiczales G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An overview of AspectJ. In: Proceedings of ECOOP, Lecture Notes in Computer Science, Vol. **2072**, Springer, 2001, 327–353.

[10] Klieber W, M. Granitzer. Using Ontologies For Software Documentation. In: Proceedings of Malaysian Joint Conference on Artificial Intelligence, 2009. `http://know-center.at/download_extern/papers/MJCAI2009 software ontology.pdf`

[11] Klieber W., V. Sabol, M. Muhr, R. Kern, M. Granitzer. Knowledge discovery using the knowminer framework. In: Proceedings of IADIS International Conference Information Systems, 2009, 307–314.
`http://know-center.at/download_extern/papers/IADIS_Knowminer.pdf`

[12] Majchrzak A., C. Wagner, D. Yates. Corporate wiki users: results of a survey. In: Proceedings of the 2006 international symposium on Wikis, ACM, 2006, 99–104.
`http://portal.acm.org/citation.cfm?id=1149453.1149472`

[13] Maskeri G., S. Sarkar, K. Heafield. Mining business topics in source code using latent dirichlet allocation. In: Proceedings of the 1st conference on India software engineering conference, ACM; 2008, 113–120.
`http://portal.acm.org/citation.cfm?id=1342234`

[14] De Medeiros A. K., B. F. van. Dongen et al. Process mining: extending the a-algorithm to mine short loops. Beta Working Paper, Eindhoven University of Technology, The Netherlands, 2004.

[15] Mogul J. Emergent (mis)behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, **40** (2006), No 4, 293–304.
`http://portal.acm.org/citation.cfm?doid=1218063.1217964`

[16] Rozinat A., R. Mans, M. Song, W. Vanderaalst. Discovering simulation models. *Information Systems*, **34** (2008), No 3, 305–327.
`http://linkinghub.elsevier.com/retrieve/pii/S0306437908000690`

[17] Schwaber K. Agile Project Management with Scrum. ISBN 9780735619937, Microsoft Press, 2004.

[18] Strohmaier M., M. Wermelinger, Y. Yu. Using Network Properties to Study Congruence of Software Dependencies and Maintenance Activities in Eclipse. 2nd International Workshop on Socio-Technical Congruence STC'09, 2009. http://kmi.tugraz.at/staff/markus/documents/2009_STC09-Socio-technical-congruence.pdf

[19] T. Xie, S. Thummalapenta, D. Lo, C. Liu. Data Mining for Software Engineering. *Computer,* **42** (2009), No 8, 55–62. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5197425

Werner Klieber
Know-Center
Inffeldgasse 21a
8010 Graz, Austria
e-mail: wklieber@know-center.at

Mansuet Gaisbauer
Hyperwave AG
Arche Noah 9
8020 Graz, Austria
e-mail: mgais@hyperwave.com

Michael Granitzer, Klaus Tochtermann
Know-Center
Knowledge Management Institute
Graz University of Technology
Inffeldgasse 21a
8010 Graz, Austria
e-mail: mgrani@know-center.at
e-mail: ktochter@know-center.at

Received March 1, 2010
Final Accepted April 29, 2010