University of Zurich
Department of Informatics

*Giacomo Ghezzi*
*Harald C. Gall*

# SOFAS Architecture

January 2010

TECHNICAL REPORT – No. 2010.IFI-2010.0002

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

G. Ghezzi
H. Gall:
Technical Report No. 2010.IFI-2010.0002, January 2010
Software Evolution and Architecture Lab
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://seal.ifi.uzh.ch

# [Technical Report]
# *SOFAS* Architecture

Giacomo Ghezzi and Harald C. Gall
*s.e.a.l. – software evolution and architecture lab*
University of Zurich, Department of Informatics
http://seal.ifi.uzh.ch/
{ghezzi,gall}@ifi.uzh.ch

January 11, 2010

## 1 Introduction

Software, in order to be successful or even just to survive, needs to be continuously maintained and adapted to evolving and/or new requirements, technologies, user needs, etc. The problem is that, as the software matures, all these additions and modifications add up, increasing the amount of disorder and thus leading to a slow but continuous deterioration in quality and usability. This makes software a very particular and unique engineering product. It is immune from most of the physical laws affecting the development and the life of any other engineering artifact, but it suffer tremendously one: Entropy (also known as "software rot"). The constant changes, so important for the survival of a software, are at the same time one of the main causes of its dismissal. Having an always up to date and thorough view of a software system, its health and its history greatly helps in avoiding this problem—or at least keeping it under control—driving its evolution in a controlled and secure way. Historical data stored into repositories by systems such as version control, bug and issue tracking, mailing lists, etc. is crucial for that purpose. Until recently, data about software development had been primarily used for historical record supporting activities, such as retrieving old versions of the source code or examining the status of a defect. Studies using this data to analyze various aspects of software development (*e.g.* software design/architecture, development process and dynamics, etc.), have emerged and flourished only in the last decade. These studies have highlighted the value of collecting and analyzing this data. Yet, each of these studies has built its own methodologies and tools to extract, organize and utilize such data to perform their research. As a consequence, easy and straight forward synergies between these analyses/tools rarely exist due to their stand-alone nature, their platform dependence, their different input and output formats, and the variety of systems to analyze. Therefore, despite this richness, we still lack ways to effectively and systematically share, integrate and study data coming from different analyses and providers. We claim that this is vital for the maturing of the field and to expand its role in supporting software development practices.

In order to tackle these problems we proposed a lightweight, flexible architecture called *SOFAS* (SOFtware Analysis Services)[6, 7]. Its goal is to offer a distributed and collaborative software analysis services to allow for interoperability of analysis tools across platform, geographical and organizational boundaries. Such tools are categorized in our software analysis taxonomy, they have to adhere to specific meta-models and ontologies and offer a common service interface that enables their composite use over the Internet. These distributed analysis services are accessible through an incrementally augmented software analysis catalog, where organizations and research groups can register and share their tools. The main purpose of *SOFAS* is thus to offer a single entry point to all the most prominent software analyses. From the catalog any potential project stakeholder can pick the analyses needed, compose them as required and run them, thereby gaining a deeper understanding of the system and its properties.

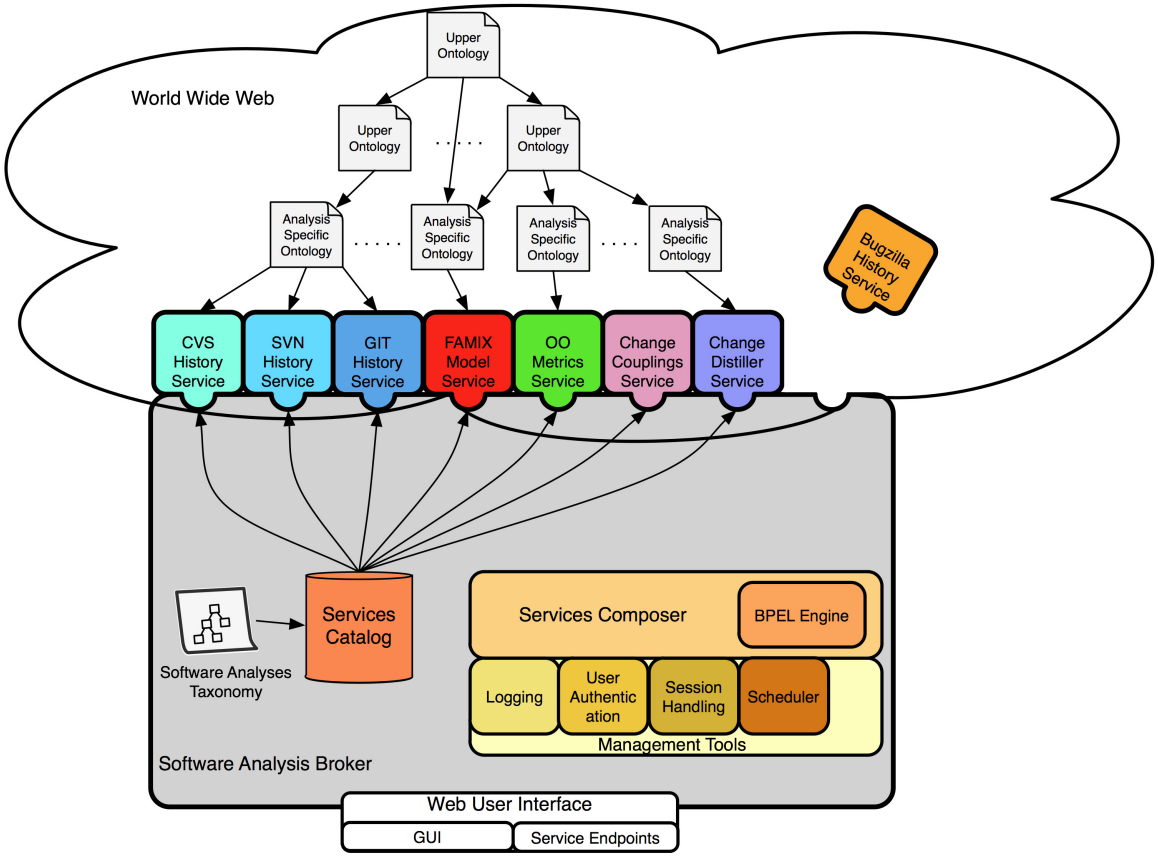


Figure 1: *SOFAS* overall architecture

This technical report explains in detail the main components of this architecture and the technologies used (details which, due to space limitations, were often left out in the cited papers).

Figure 1 gives an overall view of *SOFAS* and its three main constituents: software analysis web services, a *Software Analysis Broker*, and ontologies. Software analysis web services "wrap" already existing analysis tools exposing their functionalities and data through a web service. The *Software Analysis Broker* acts as the services manager and the interface between them and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. Ontologies are used to define and represent the data consumed and produced by the different services. Upper ontologies define more generic concepts common to several specific ontologies; thus, they provide semantic links between them. For that we developed a first corpus of ontologies called *SEON* (Software Engineering ONtologies). In Section 2, we discuss why we decided to use web services and we then introduce all the existing analysis services we developed. In Section 3 we present the *Software Analysis Broker* and its main components. In Section 4, we discuss in detail *SEON* and all its ontologies. We then finish with a short usage example, to give a flavor on how *SOFAS* works given a specific task, from a user perspective.

## 2 Software Analysis Services

We decided to use web services over other competing middleware technologies as it is a well-known standard and already offers many of the features we need: language, platform and location independence and service composition. Independence is achieved with the use of XML-based

languages to describe the services (WSDL[1]) and a simple, lightweight communication protocol (SOAP[2]) intended for exchanging structured information, formatted into XML-based messages, in a decentralized, distributed environment, normally using HTTP/HTTPS. These characteristics of loose coupling, published interfaces and a standard communication model enable existing applications to easily expose their functionalities through web services. The internal logic, the input and output formats used, the platform and language under which the original tool runs remain the same but are hidden behind the web service. This is extremely important for the *SOFAS* platform because in this way we are able to reuse the analyses we already had with no need of adaptation and reengineering. Moreover, since no code, but just the plain results, are actually exchanged, the intellectual property contained in each analysis is also protected.

Web services, because of the verbose XML format, can in general be considerably slower than such as CORBA, DCOM and ICE. However they have three big advantages, especially for our purpose. First, the are way less complex and hard to understand then the other solutions. Second, a well known, standardized composition and orchestration language for them already exists: BPEL4WS (Business Process Execution Language for Web Services)[3], an XML-based language designed to enable task sharing for a distributed computing—even across multiple organizations—using a combination of Web services. Third, SOAP uses HTTP/HTTPS, which allows for easier communication through proxies and firewalls, a must for us, as we want to easily overcome as many limitations and boundaries as possible.

WSDL specifies a standard way to describe web services, the structure of their input and output and how to invoke them at a syntactic level. However, there is no way to programmatically know from the bare description what a service actually offers and what the data it consumes/produces means. A human user could only get that information thanks to some written description or knowledge shared somehow by the service creator. Users without that type of information, be it humans or, more importantly, machines cannot figure it out that by themselves. Moreover, each service would then still structure its results according to its specific format and follow its own metamodel. Thus, the integration and combination of results would be possible only with cumbersome manual ad-hoc data preparation and transformation. A common exchange format providing a rigorous, unambiguous syntax and semantic of data is indispensable. Several researchers have pushed for common interchange formats such as GXL (Graph eXchange Language) or XMI, but their efforts have remained largely unheard. Furthermore, the existing exchange formats focus only on the syntax of data, but do not address its semantic at all. We address this problem by using semantic web technologies, in particular OWL[4] and SAWSDL[5] (Semantic Annotations for WSDL and XML Schema). In our opinion, these technologies have a lot of potential in the area of software engineering in general, and more specifically in software evolution analysis. What makes them really worthwhile using is that they help tackle both problems, i.e. meaningful service description and data representation, at once.

With OWL we are able to give output and input data a clear, univoque semantic (its original purpose), but also a precise syntax, as it is a standardized, XML based language. The fact that it is XML based also gives us a sound and well known data format to use and the ability to share that data between different types of computers using different types of operating system and application languages. Furthermore, the properties related to its ontological nature make it really stand out from all the other already existing solutions:

1. Heterogenous domain ontologies can be semantically "linked" to each other by means of one or more upper ontology, describing general concepts across a wide range of domains. In this way it is possible to reach interoperability between a large number of ontologies accessible "under" some upper ontology. In terms of software analysis services, it means that results from the most disparate types could be automatically combined given that they share some

[1] http://www.w3.org/TR/wsdl
[2] http://www.w3.org/TR/soap12-part1/
[3] http://bpel.xml.org/
[4] http://www.w3.org/TR/owl-features/
[5] http://www.w3.org/TR/sawsdl/

common concepts. Section 4 and in particular Figure 16 show what that means in a real case, with concrete ontologies.

2. With the OWL Description Logic foundation it is possible to perform automatic reasoning and derive additional knowledge.

3. We can use a powerful query languages, such as SPARQL, or more advanced techniques, such as guided natural language[10].

4. In contrast to XML and XQuery that operate on the structure of the data, OWL treats data based on its semantics. This allows for an extension of the data model with no backwards compatibility problems with existing tools.

Using Semantic Annotations for WSDL and XML Schema, web services and ontologies can be effectively integrated together to create *semantic web services*, which are crucial parts of our approach. Semantic annotations can be attached to the most important parts of a web service definition, adding semantic meaning to it, as shown in the example in Figure 2. First the service itself can be declared to represent a particular concept of an ontology, in our case a specific analysis category; in the example, a CVS release history data extractor. Second, its inputs and outputs can be declared of being concepts of specific ontologies and thus have a specific semantic meaning. In this way we know precisely what the service returns. Moreover, with all this information we can easily check, for every new service being registered in our analyses catalog, whether it supports inputs and provide results conforming to ontologies specific to the analysis it is declared to implement (e.g. every service offering CVS extraction has to return a version control history). We will explain in detail the ontologies we developed to describe software evolution data in Section 4.

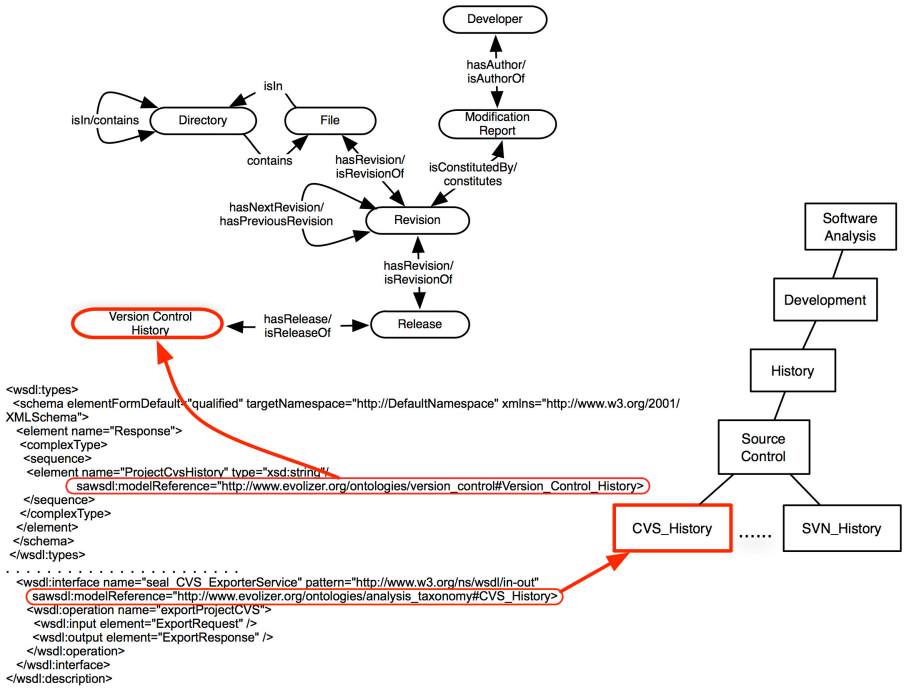


Figure 2: An example of a semantical annotated web service definition.

So far, we have developed (or we are in the process of finalizing) the analysis services shown in the architecture overview in Figure 1.

**CVS, SVN, GIT History services.** They extract the version control information comprising release, revision, and commit information from CVS, SVN or GIT repositories: who changed

when/which source file and how many lines have been inserted/deleted. In order to work, these services only need the url the repository and valid user credentials (username and password). Additional options to further fine-tune the data extraction are also available. For example, extracting the history of just a specific revision interval or fetching the content of every file revision of specific file types. This last option is particularly useful in case additional analyses need to be done on the actual source code (model extraction, metrics calculation, etc.).

**FAMIX model service.** Given just the source code of a software, it extracts its static structure in the form of a FAMIX model [6] (a language independent meta-model describing the static structure of object-oriented software). The service is able to partially reconstruct the static structure even when the source code does not compile, or has errors, by using the heuristics already developed for ZBinder [9].

**OO Metrics service.** It extracts the most important software metrics from a software system. This service accepts two types of inputs: raw source code or FAMIX models. Given this data, it calculates, for now:

- Fan-In and Fan-Out of a class, method or package.
- McCabe's cyclomatic complexity [8] of a class, method or package.
- Lines of code (LOC) of a class, method or package.
- Depth of inheritance tree (DIT) of a class.
- Number of methods overriding a method in any one of the super-classes of a class (NMO).
- Number of direct sub-classes of a class (NOC).
- Number of attributes (static and non) of a class or a package (NOA).
- Number of methods (static and non) of a class or a package (NOM).
- Number of parameters of a method (NOPAR).

If no other piece of information is given to the service, it will calculate all the metrics for all the source code entities found. Otherwise, the user can set the service to calculate only specific metrics, for specific entities.

**Change Coupling service.** Given the version control history of a software project, it extracts the change couplings for all the files. This means that for every versioned file, it extracts what other files were simultaneously changed with them, how many times and when. The more two files changed together, compared to the total number of changes they were involved with, the more they are coupled.

**Change Distiller service.** Given a project version control history, it extracts, for each revision, all the fine-grained source code changes of each source code file. These changes are then classified following the change types taxonomy proposed in [3]. The algorithms used to extract these changes are based and the ones developed by Fluri et al. [4] for the original Change Distiller tool [5].

**Bugzilla History service.** It extracts all the issue tracking historical information (problem reports and change requests) from a given Bugzilla repository. This data is usually used as-is or together with the project version control information. In the first case, it can help assess the average bug-fixing time, the distribution of bug severity, etc. In the second case, it can be used for more complex analyses, such as location of fault prone files, location and analysis of bug fixing changes, bug prediction, etc. As for the version control services, also this one can be set to import just a range of issues, instead of the whole history.

Note that all these services structure the data extracted following specific ontologies, which will be explained in Section 4.

[6] http://www.moosetechnology.org//docs/famix

# 3 *Software Analysis Broker*

The *Software Analysis Broker* takes care of keeping track of the different analysis services, classify them in a registry, query, monitor and coordinate them. It basically acts as a "layer" between the services and the users. In this way, the user does not have to interact directly with the raw services, but can take advantage of several additional utilities facilitating the whole experience. The *Software Analysis Broker* is made up of four main components: the *Services Catalog*, a series of management tools, the *Services Composer* and a user interface.

## 3.1 *Software Analysis Broker* User Interface

The user interface is the access point to all the functionality the *Software Analysis Broker* offers. It consists of a web GUI meant for human users and a series of web service endpoints to be (semi)-automatically used by other applications, without the need of human intervention. Through this interface the user can browse through the *Services Catalog* to check the analyses offered and select some of them; or she can query it to get a specific analysis right away. Figure 3 offers a sketch of it. Apart from the catalog, the user can also pick from some already predefined combinations of those services into high level evolution analyses workflows we call *analysis blueprints*, so she does not have necessary to fiddle around to find the right services and/or composing them. For example:

1. a workflow that, given a project version control system repository it extracts for each of its releases, a wide range of source code metrics.

2. a workflow that, given a project version control system repository and its bug tracking system, it extracts for all the files in each release the related bugs and classifies them according to their number and gravity.

Once the desired services, or predefined analysis blueprints, are found and selected, the user needs to set some service-specific settings. For example, a service extracting the release history from an SVN repository, would need the URL of the repository itself and, possibly a username and password to access it. We introduced some of them already in Section 2. Moreover, if the user chose to combine two or more services into a workflow, she would need to actually define how to do that. That is, what their sequence is, what output of a service should be fed as input to another service, etc. The user interface offers an intuitive, high level way to do that, allowing the user to combine the services in a "pipe and filter" fashion with just boxes and arrows. The real composition of those services into a BPEL workflow and its execution is then taken care of by the *Services Composer*. In this way, the user does not even have to know about BPEL and its syntax at all.

## 3.2 *Services Catalog*

The *Services Catalog* stores and classifies all the registered analysis services in a clear and unambiguous way. In this way, any user can automatically discover services needed, invoke them and then fetch the results. To do that, a clear and unambiguous classification is essential. Based on the existing software analysis techniques, we developed a specific software analysis taxonomy to systematically classify the existing and future services. Figure 4 shows a concise version of it. This taxonomy divides the possible analyses into three main categories based on their focus: the development process, the underlying models or the actual source code.

*Software development analyses*, as shown in Figure 5, are subdivided into those targeting the development history (extraction, prediction and analysis of source code changes and bugs), its underlying process, and the teams involved in it (their dynamics and metrics). *Model analyses* include those targeting the extraction, either dynamic or static, of specific behavioral and structural model representations (UML, FAMIX, call graphs, etc.) and those computing differences between two models, usually of two different versions of the same system. Figure 6 shows a compact view

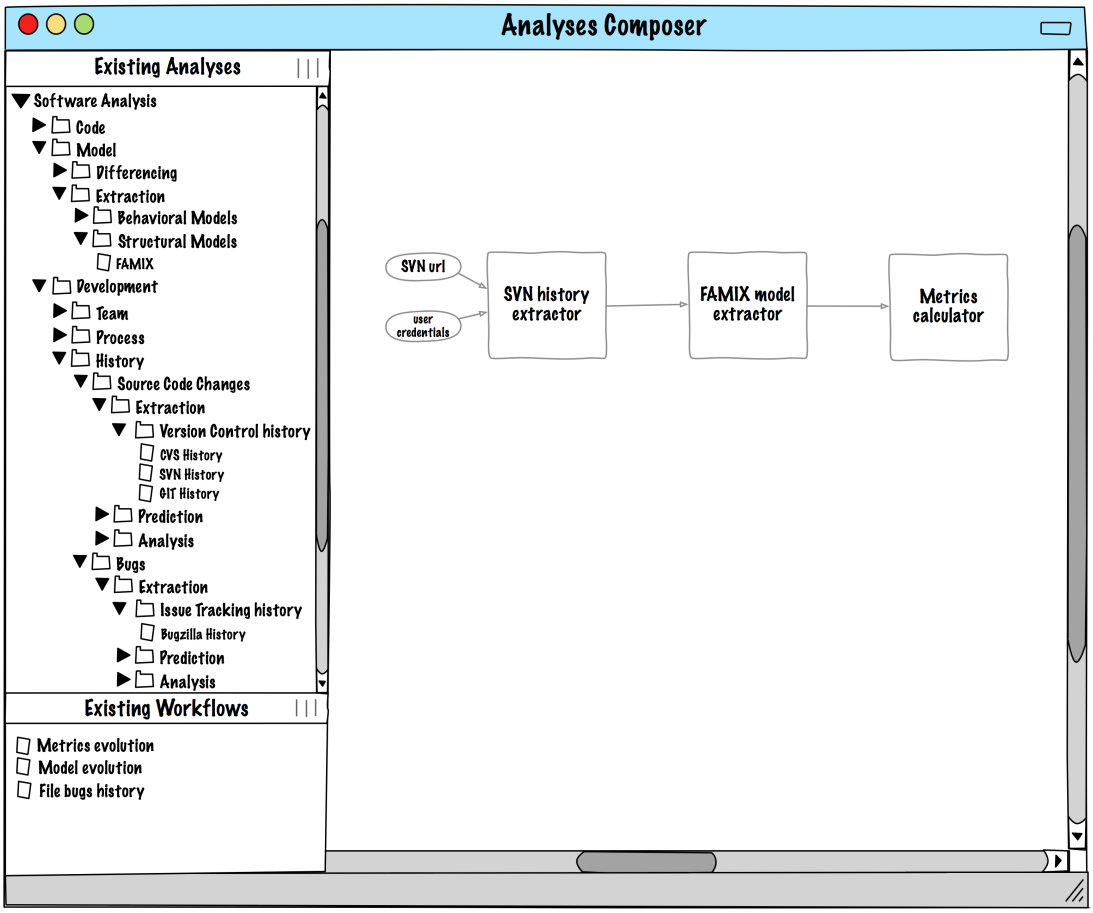


Figure 3: A sketch of the *Software Analysis Broker* GUI.

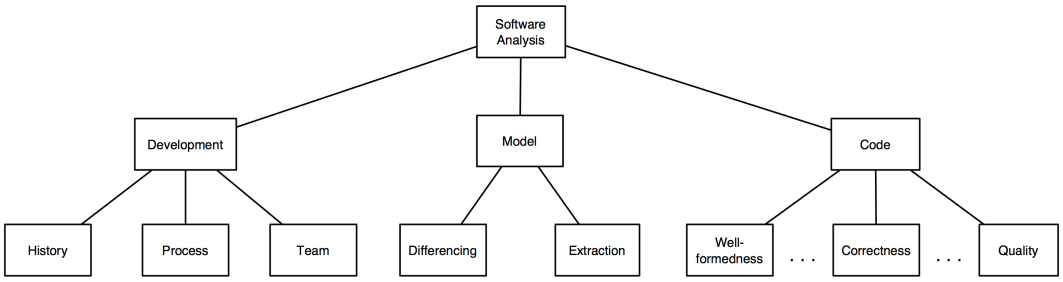


Figure 4: A condensed view of the software analysis taxonomy

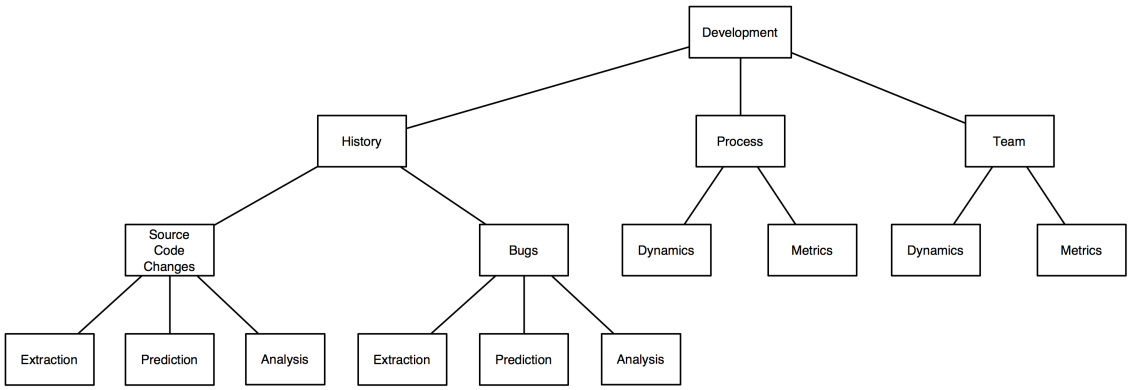


Figure 5: A view of the software development branch of the analysis taxonomy

of this part of the taxonomy. *Code analyses* are further divided into many categories such as checking code well-formedness, correctness and quality. For example, the code quality category is then further divided into subcategories dealing with code security, conciseness, performance and design. The latter category contains, among others, extractors and analyzers of design metrics and code-smells, as shown in Figure 7.

The taxonomy is defined as an OWL[7] ontology. Thus the catalog itself ends up being an instance of that ontology and every registered service an instance of a specific class of that ontology. The ontology is managed and stored in a triplestore and accessed using JENA[8], an open source framework meant exactly to allow for the querying, storing and analysis of RDF/OWL data through a high-level, intuitive API. We decided to develop this lightweight semantic web-based custom solution, instead of using UDDI[9], the standard solution for web service registries, for several reasons. The most prominent are related on how it deals with taxonomies, how they are defined, how they are used to classify and then fetch services. In fact, UDDI's taxonomies are usually rather simple, flat and with a convoluted—often unreadable—definition, especially compared to the cleanness and richness one can reach by using OWL. This obviously highly affects the quality and broadness of classification and subsequent querying of services. On the other hand, with OWL the classification can be as complex and specific as we want the taxonomy to be. Moreover powerful query languages, such as SPARQL[10] can be used to query the catalog and fetch specific services. With these languages, the querying options become manifold: services can then be queried based on what categories they belong to, on any of their attributes, on the attributes of any of the categories they belong to, etc.

## 3.3 Management tools

Most of the times, just calling services or combining them is not enough. In particular, this holds for long running, asynchronous web services. They need, for example, to be logged and monitored to check if they are up and running, if they are in an erroneous state and why, if they have have completed a required operation, etc. Otherwise a consumer might end up waiting for a particular operation without knowing why and how it failed. Even though these functionalities are vital for end users, their use should be as transparent, standardized and automated as possible. Thus, we implemented a series of services that take care of implementing that as services. As a result, calls to them can be easily and automatically weaved into a user defined workflow, even if only just one service was chosen. The *Services Composer* takes care of doing that.

[7] http://www.w3.org/TR/owl-features/
[8] http://jena.sourceforge.net/
[9] http://uddi.xml.org/
[10] http://www.w3.org/TR/rdf-sparql-query/

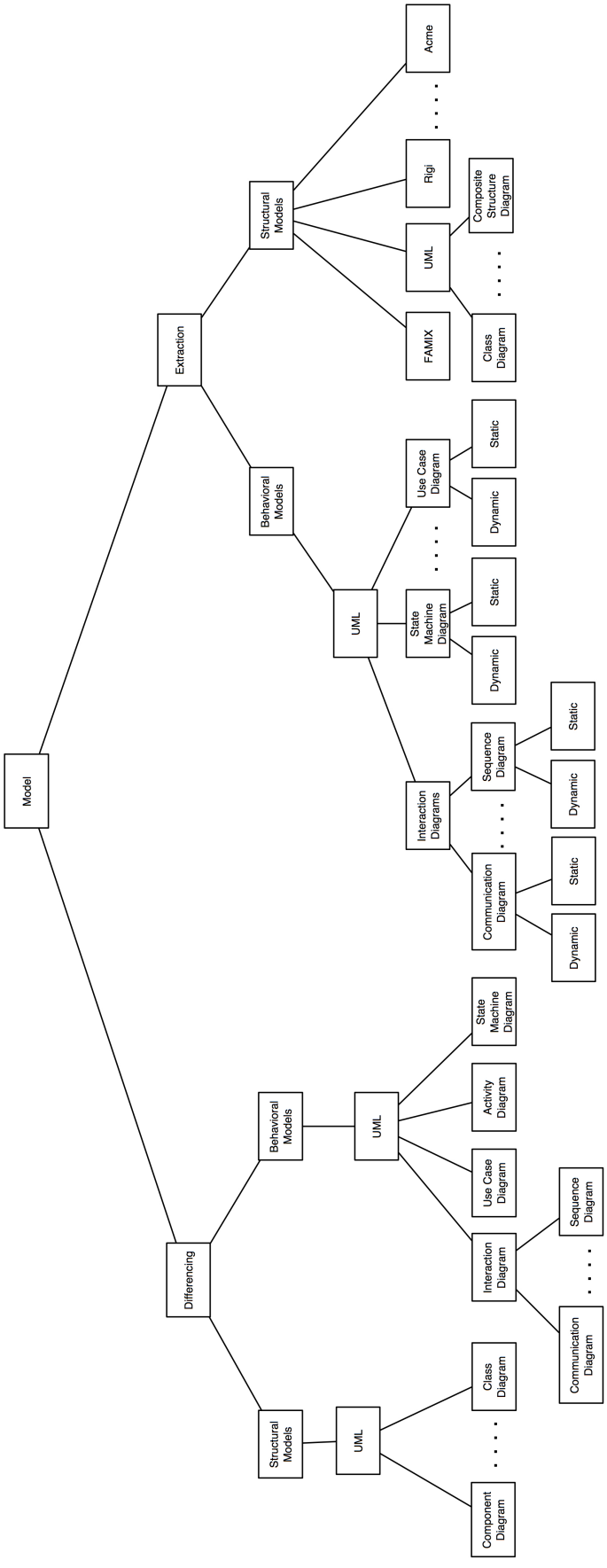


Figure 6: A view of the software model branch of the analysis taxonomy

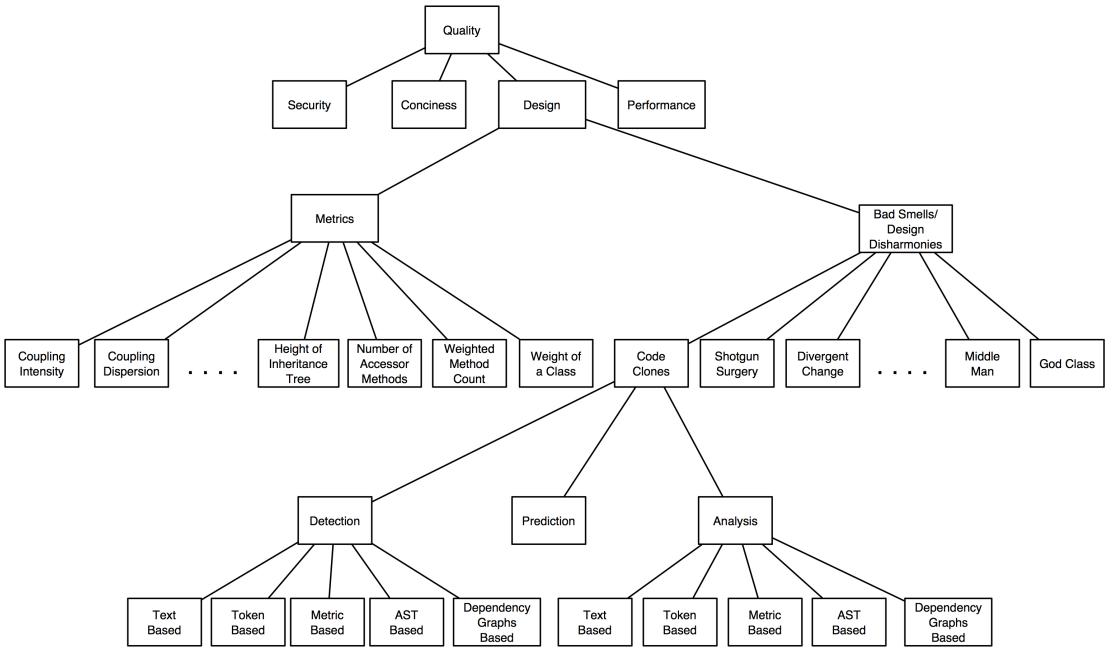


Figure 7: A view of the software design quality branch of the analysis taxonomy

### 3.4 *Services Composer*

This component works both as an interpreter and as a BPEL engine. In fact, it translates the high level service composition workflow defined by a user through the UI into a real, executable BPEL process and executes it using a standard BPEL engine. The decoupling between the user composition definition and the actual composition language is useful for two reasons. First, it allows the user to compose services in a intuitive way, hiding the complexity and technicalities of BPEL. Second, calls to additional services can be automatically weaved into a user defined workflow. In our case, the *Services Composer* adds calls to important management services such as logging, user authentication, error handling, etc. that allow the workflow to run smoothly and effectively and that the user should not have to care about.

## 4 *SEON*

The goal of our Software Engineering Ontologies, *SEON*, is to provide a structured and integrated corpus of ontologies giving an adequate view on the different concepts of software evolution and software engineering in general. In the past few years, researchers have started to address the use of ontologies in software engineering. Several ontologies have been designed to represent data in this domain. However all these ontologies covered just specific parts of the domain. We *SEON* we intend to overcome this deficiency.

So far we have developed ontologies to describe version control data, issue tracking data, static source code information, change coupling and software design metrics. For each of the first three subdomains we have developed higher level ontologies defining all the common concepts (*e.g.* the concepts of releases and revisions are common to any existing version control system). For systems specific or language dependent concepts that could not be put into those generic ontologies we developed some concrete low-level ontologies: (1) Bugzilla and Trac ontologies in the case of issue tracking; (2) CVS, SVN and GIT ontologies for version control; and (3) Java and C# for static source code information. Figure 8 depicts the basic structure of *SEON*. The three major subdomains are represented as individual ontology pyramids. The shared concepts and properties are defined by the ontologies at the top, while the system/language specific ones inheriting all that

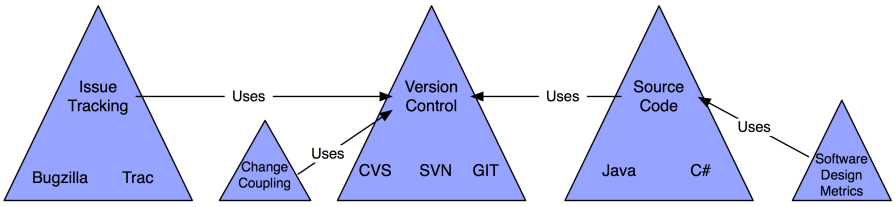


Figure 8: *SEON* overall structure

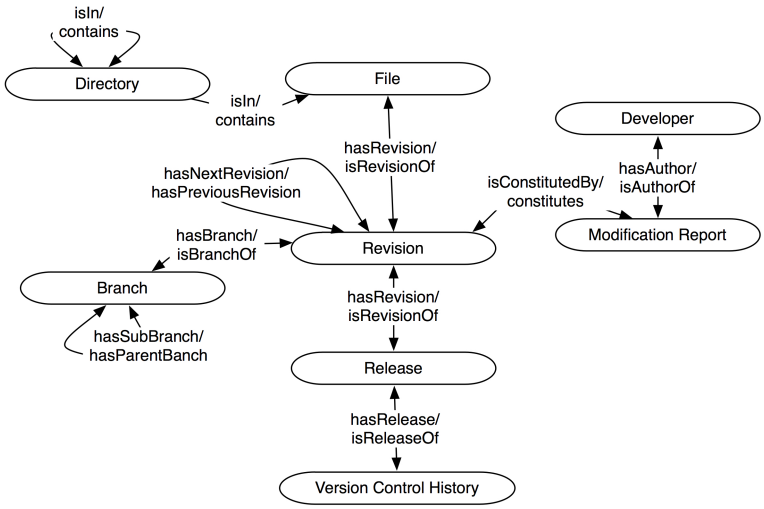


Figure 9: Overview of the version control history ontology.

are at the bottom. Moreover the source code, issue tracking and change coupling ontologies import properties from the version control system one, as the metrics ontology does from the source code one.

The version control pyramid is the core of *SEON* as it interconnects the three major subdomains we developed so far. This is not surprising as the version control history is essential for any type of software evolution analysis. Figure 9 depicts the classes and their relationships making up this ontology. All the essential concepts of version control are represented in this ontology. They are sufficient to model and describe historical data of all the major systems. In fact all of them share the same conceptual model, with just some slight, easily spottable differences in terminology. For example, they might use *baseline*, *label* or *tag* to identify a snapshot of the project. In our case, we decided to use a more neutral term as *release*. Thus, the CVS, SVN and GIT ontologies only add very few additional concepts to the generic ontology. The SVN one, as shown in Figure 10, adds the concepts of copies, moves and renames as these operations are poorly supported—or not supported at all—by others systems.

Using an issue tracking system, developers can report issues (usually bugs or feature requests) about a software project. In this case, the concepts and terminology are looser then in version control. However, no matter what system is used, the *issue* is always the central concept. Thus it is also the focal concept of the ontology. Moreover, through that, the issue tracking data is conceptually connected to version control history. In fact, most of the times, especially for bugs, an issue reports some problems about specific file revisions and, viceversa, some revisions fix specific reported issues. Figure 11 shows the classes and relationships of this ontology. Also in this case, all the essential concepts are covered in the generic ontology. All issues are filed against a specific version and component of the software by a reporter. They are then, sooner or later, assigned to a person (the assignee) who would take care of it. Associated to every issue are also all the possible discussions and comments the assignee, the reporter and other people interested

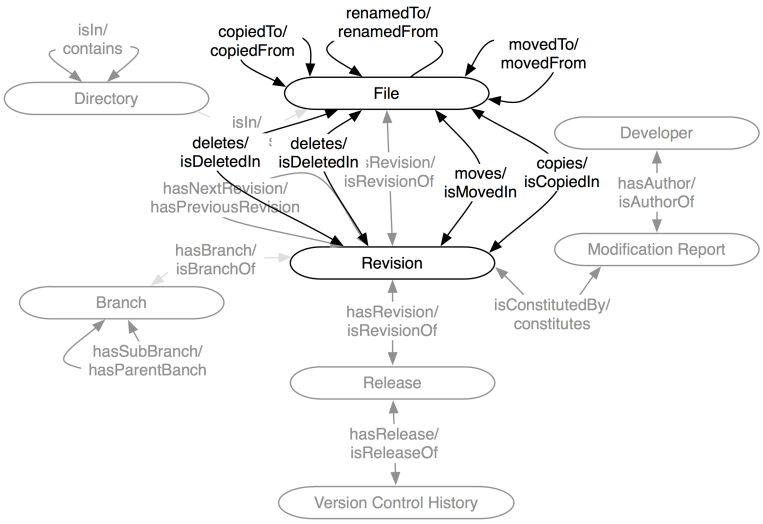


Figure 10: Properties of the SVN version control history ontology (the generic ontology is in grey).

in the issue (a.k.a. CC persons) might have about it. The major differences lie in how issues are classified, prioritized and how richly they are described. For example, Trac classifies issues in three categories (defect, enhancement and task), while Bugzilla only has one (enhancement and tasks are just considered bugs with low severity). Moreover Bugzilla has a slightly richer issue description than Trac. For example, it also keeps track of OS and hardware under which the issue was experienced. These differences are shown in the Figures 12 and 13. As of now, the Trac ontology is not yet used, as no services to extract its history are developed yet.

The source code ontology models all the static source code structures based on the FAMIX meta-model. We decided to use FAMIX instead of other meta-models, such as UML, as it has a finer granularity and higher level of details. FAMIX was already devised as a language independent source code model for OO programming languages, therefore we were able to represent all the important concepts in the generic ontology. We only created the Java and C# ontologies just to address the few particularities, and to be able, when needed, to be as specific as possible (*e.g.* the two languages do not have the same set of primitive data types). However, as of know, in our FAMIX service we only use the generic one, as it fulfills all our needs. In fact, all the concepts that are vital to understand a system, analyze it and reconstruct its architecture are all common to any OO language and are thus represented by the generic ontology. The central concept of this ontology is the *class*. Through a class the source code ontology can be linked to a version control history as classes are contained in versioned files. Figure 14, depicts the overall structure of this ontology.

The metrics and change coupling ontologies are simple as they describe rather basic and unstructured data. The first one classifies most used software product metrics as described in many works, such as [2, 8]. All these metrics are either calculated at class or method level, so the concept of metrics itself is associated to the concept of *entity* of the source code ontology, which represent both classes and methods. These metrics either measure Object Oriented-specific properties (*e.g.* class cohesion, number of methods in a class, etc.) or attributes common to any programming language. The latter are further divided in the ones addressing size (*e.g.* lines of code, number of statements, etc.) or procedure properties (*e.g.* cyclomatic complexity, fan-in/fan out, etc.). Figure 15 shows a condensed view of this ontology.

The change coupling ontology describes how well two files are coupled in a project: how many times and when they were changed together during the lifetime of a software. This last ontology, along with all the other ones and their relationships, is shown in Figure 16. For more additional and specific details about the single ontologies and all their properties, we refer to [1].

*SEON* is still being improved and by no means complete. We envision many other ontologies

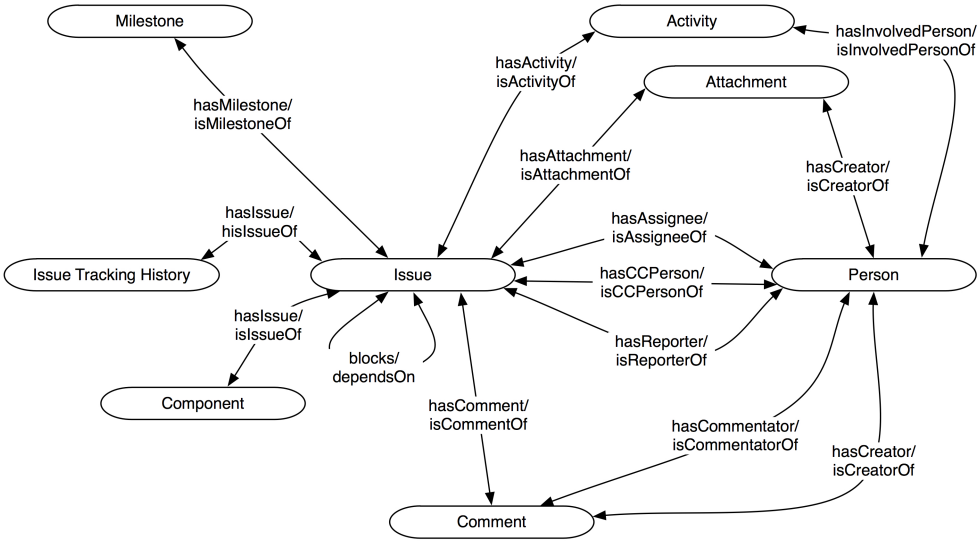


Figure 11: Overview of the issue tracking ontology.

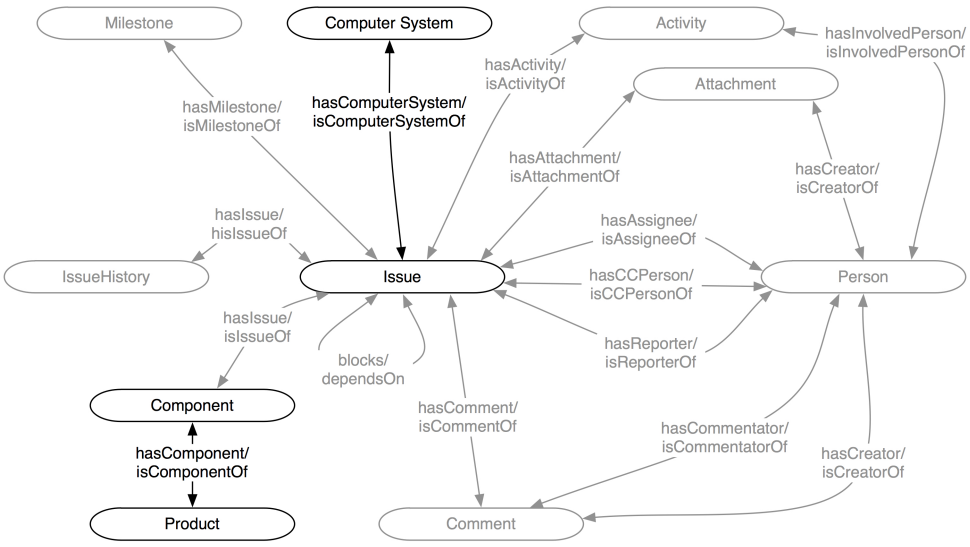


Figure 12: Overview of the Bugzilla tracking ontology (the generic ontology is in grey).

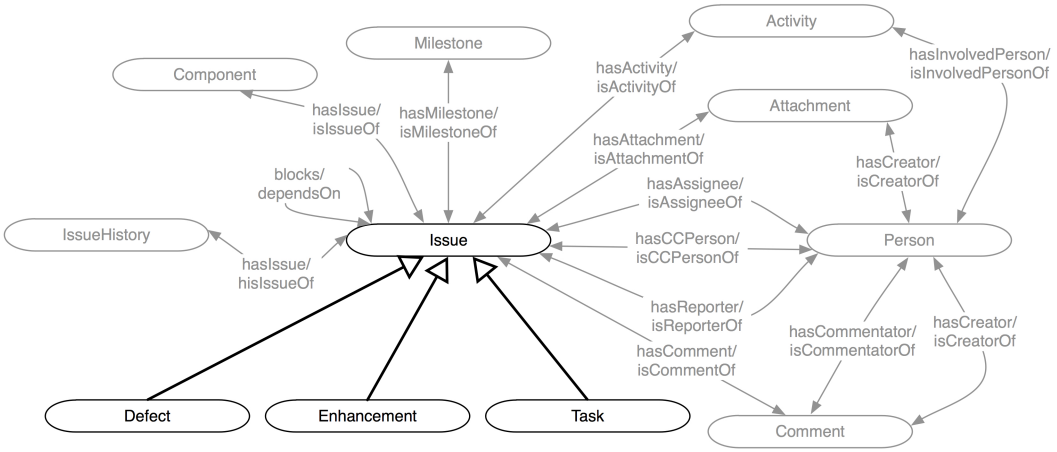


Figure 13: Overview of the Trac tracking ontology (the generic ontology is in grey).

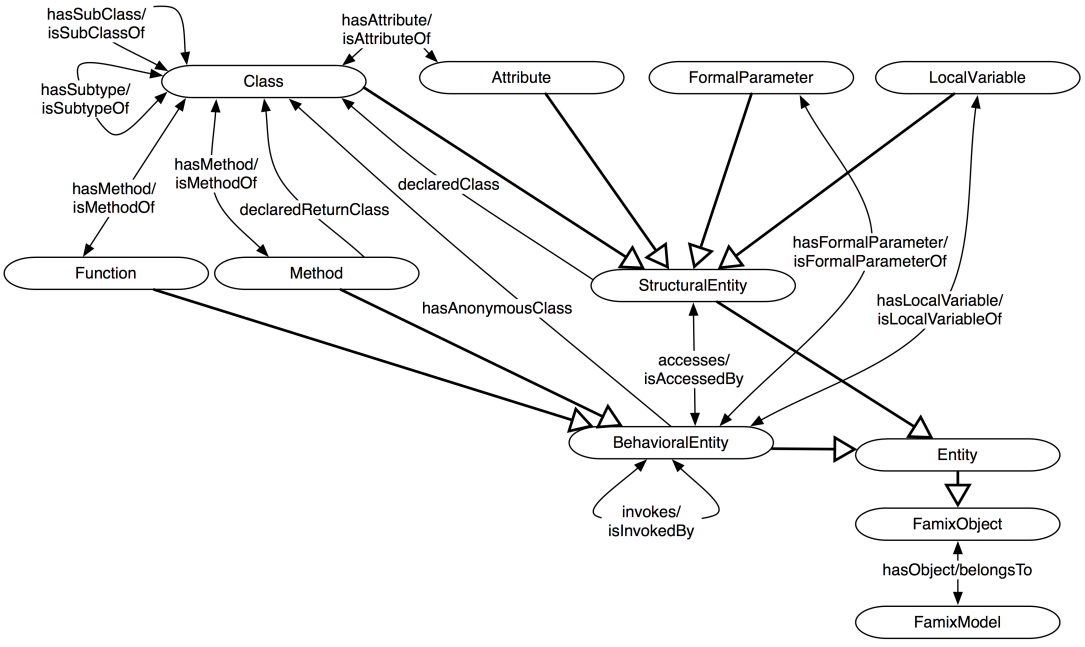


Figure 14: Overview of the FAMIX ontology.

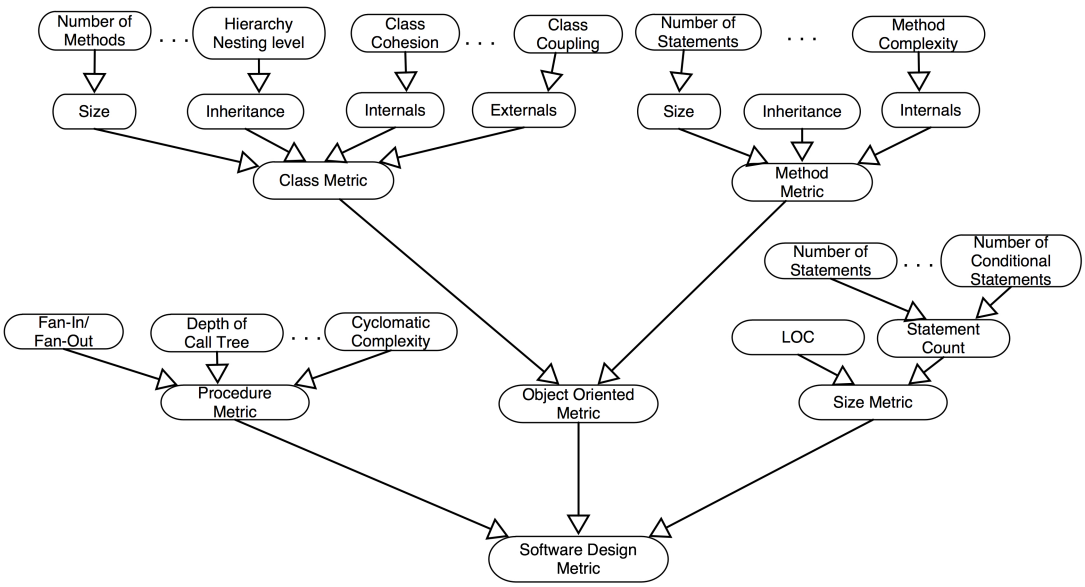


Figure 15: Overview of the metrics ontology.

to incrementally added by us or others. For example, ontologies targeting other code quality measurements (*e.g.* code clones or code smells) or source code change types (as defined in [4]). Moreover, many supplementary ones describing other version control systems (*e.g.* Mercurial), issue tracking systems (*e.g.* Jira, Mantis) and programming languages (*e.g.* C++, Eiffel, Python). Basically, any additional ontology can be added without influencing or changing the definition of the already existing ones.

# 5 A usage example

Let us see how *SOFAS* works if a user wants to calculate a series of OO metrics for every release of a project, in order to see how they evolved. This can be used, for example, to assess how the overall quality of the software progressed and spot suspicious parts that might need refactoring.

1. *Providing the URL of a projects code repository*: To start, the user has to have some initial data about the project to analyze: *e.g.* the URL to the project's source code repository.

2. *Searching for appropriate software analyses*: Then, by navigating the *Services Catalog* via its user interface, the user needs to find the appropriate analysis service supporting history extraction for the specific revision control system.

3. *Selecting services*: From then on, the user is guided into the composition of the first chosen service with additional ones. Once a service is selected, the system knows right away what data it produces and can therefore suggest all the possible services that can consume that data. This is possible thanks to the use of semantical annotation in the service description as we can describe what services input and output really mean (syntactically and semantically).

4. *Using services*: In this case, once the revision control history extractor is selected, the system, knowing that it produces a "Source_Control_History" as shown in Figure 2, picks all the services consuming it: Change Coupling, Change Distiller and FAMIX Model.

5. *Concatenating services*: As before, all the services that could possibly additionally be concatenated to the ones already selected are presented to the user. This iterative process goes on until the user has fulfilled her needs. In our example, the user just concatenates to the

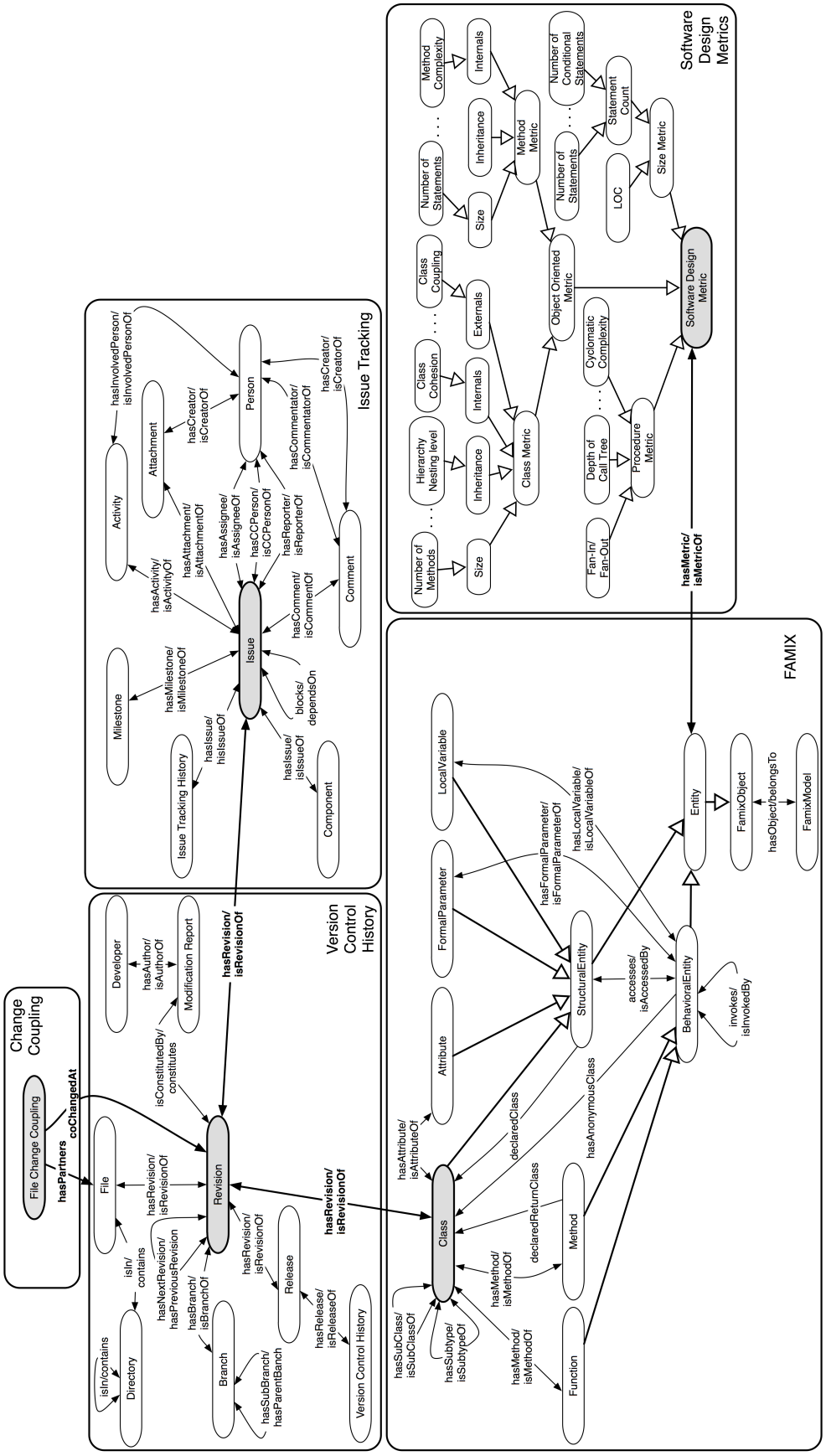


Figure 16: Overall view of the *SEON* ontologies.

FAMIX Model and the OO Metrics services, which calculates OO metrics of a software given its FAMIX model.

6. *Setting additional service options*: In some case, the user can set additional service options. For example, calculate just some specific metrics, instead of the complete set.

7. *Translating into BPEL workflow and executing services*: Eventually, when the workflow definition is completed, it is passed to the *Services Composer*, which translates it into a proper BPEL workflow, weaves in the calls to all the needed management services, executes it and notifies the user when the job is done.

# References

[1] J. Belik. SEON, The Software Engineering Ontology. Master's thesis, University of Zurich, Department of Informatics, Zurich, Switzerland, 2009.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[3] B. Fluri and H. C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 9th International Conference on Program Comprehension*, pages 35–45. IEEE Computer Society, June 2006.

[4] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[5] H. C. Gall, B. Fluri, and M. Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, January/February 2009.

[6] G. Ghezzi and H. C. Gall. Towards Software Analysis as a Service. In *Proceedings of Evol'08, the 4th Intl. ERCIM Workshop on Software Evolution and Evolvability at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering*, L'Aquila, Italy, September 2008. IEEE Computer Society.

[7] G. Ghezzi and H. C. Gall. SOFAS: Software Analysis as a Service. *IEEE Software*, 26(4):to appear, July/August 2010.

[8] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[9] M. Pinzger, E. Giger, and H. C. Gall. Handling unresolved method bindings in eclipse. Technical report, Department of Informatics, University of Zurich, Switzerland, 2007.

[10] M. Wuersch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting Developers with Natural Language Queries. In *Proceedings of 32nd International Conference on Software Engineering*, page to appear. IEEE Computer Society, May 2010.