Acquiring software semantics for retrieval of software components on the web

Marta Sabou

Vrije Universiteit Amsterdam

Abstract

While the number of software components increase on the web, finding these components and dynamically integrating the active ones is very difficult. Extending Semantic Web technology to describing these components seems to be a solution. However, the easy acquisition of descriptions for software components is an open issue.

We focus on facilitating the acquisition of software semantics. To perform this, we (1) determine a set of resources that contain semantics about software and (2) propose methods to extract these semantics. Finally, to validate the quality of the semantics (3) we will use it to lay the foundations of a semantic search engine for software components on the web.

Contents

1 Introduction

2 Background and motivation

2.1 The Semantic Web

2.2 Software on the web

2.3 Semantics of software

2.3.1 Traditional approaches

2.3.2 Software descriptions on the Web

2.3.3 Comparison

3 Research statement

4 Concrete plans

4.1 Envisioned research strategy

4.2 Open Issues

4.3 Planning
1 Introduction

The aim of this document is to state the focus of our research interest. It is structured in three parts.

Our work is influenced by two major phenomena: the Semantic Web (2.1) and the growing interest for web-services (2.2). After we shortly describe these two fields we conclude that there is a need for semantically describing software components. Section 2.3 elaborates on this by presenting traditional (2.3.1) and novel techniques (2.3.2) to describe software and concluding in 2.3.3 that novel techniques have better applicability for the web.

The statement of our research question (3) follows naturally and is highly motivated by the analysis performed in chapter 2.

The last part of this document contains some concrete ideas about the working strategy (3.1) as well as about some issues that still need to be decided (3.2). In the light of these observations we propose a working plan for the upcoming three years (3.3).

2 Background and motivation

As any new invention the Web does not only bring benefits but also challenges as it fundamentally differs from any traditional information space. First, the web is larger then any repository or knowledge base. Second, we remark the diversity of existing information as well as its high change rate. Also, these enormous repository is spread over millions of web-hosts. Finally, we conclude that many of these features are influenced by the fact that the web is an open environment: anybody can publish anything at anytime [19]. These extreme features (size, diversity, change rate, distribution) make the web a great playground for testing and extending traditional methods in many research fields such as Knowledge Based Systems, Agent Technology or Software Libraries, to name a few.

Our work also relates to the web and addresses the intersection of two interesting phenomena which promise to revolutionize the current web.

2.1 The Semantic Web

The major problem on the web is discovering relevant information and combining this information in a relevant way. Current search engines, based on keyword search, are of little support. While they have a high recall, retrieving all the pages that contain a certain keyword, they have a very low precision, i.e. it is very difficult to retrieve the pages that are relevant for the users’ request. Nowadays we have to deal with limited support for retrieving web-pages, and no support in integrating the information existent on the web.

Fuelled by this problems, considerable research was done in extending the web with machine understandable semantics. Such a Semantic Web allows machines to better perform tasks such as searching and integrating information. This technology triggers a transition from the current human-understandable web to a formal
The idea of the Semantic Web is to augment each web-resource with a formal description that can be understood and used by computers to perform tasks. The content of the descriptions is expressed in a formal language using terms that are agreed by a large community (in ontologies). Schematically this is represented in figure 2.1:

![Diagram](image.png)

Figure 1: The idea of the Semantic Web

Several issues need to be taken into account when building a description: (1) the content that needs to be expressed, (2) the ontology that provides the terms to express the content and (3) the formal language in which both the ontology and the descriptions are written down.

- **Content** While a lot of things can be stated about a certain resource, e.g. a web-page, the formal description should provide the content that is necessary to support a certain task.

- **Ontology** A vital role in the Semantic Web is played by ontologies. Ontologies are domain conceptualisations with the important features that they are (1) agreed upon by multiple parties and (2) expressed in a formal language. An ontology describes the terms of a domain, their relationships and axioms expressing certain constraints. This knowledge is described in a formal, i.e. machine-understandable, way.

- **Language** Significant effort is devoted to develop semantic languages that are expressive enough to allow describing the large variety of existent knowledge. The first, very simple semantic language, RDF(S) is already used and has significant tool support. Other, more expressive languages were built on top of RDF(S), such as OIL and DAML, finally merged in DAML+OIL. Currently, the OWL web ontology language is developed, based on the experiences with DAML+OIL and on industry demands.

The problem of the Semantic Web: How to get the semantics? Even if the idea of the SW is simple, it has a weak point: it crucially depends on the existence of ontologies and annotations. However, the sheer size of the web
and its incredible diversity prohibits manually creating domain ontologies and annotations. Several researchers addressed this issue and proposed a variety of methods to (semi-) automatically extract ontologies and to support annotations.

**Some of the solutions to acquiring semantics are:**

- **Ontology learning** The field of ontology learning investigates methods to (semi-) automatically extract ontologies from exiting resources. A tool that extracts light-weight ontologies (i.e. concept hierarchies) from existent database schema is described in [11]. This article also provides an overview of AI methods that can aid ontology extraction from other sources that db schema.

- **Semi-automatic HTML-page annotation** Several similar tools were built to support the process of annotating an HTML page. The basic idea is to use a concept extraction tool that parses the document to be annotated and recommends which part of the text should be linked to certain terms of an ontology. At EKAW 2002, two papers propose using the same concept extraction tool (Amilcare [4]) to support semi-automatic annotations [9, 20].

- **Automatic DAML-annotations** The AeroDAML [10] initiative attempts to automatically generate the DAML annotation of web pages. It uses NLP techniques to make sense of the text and annotates it according to existing ontologies (ex. Cyc) or custom ontologies. The tool can be also used to support "semi-automatic" creation of semantics. The basis of AeroDAML is the information retrieval module called AeroText. The intended use of the extracted semantics is information retrieval and simple question answering.

A general comment to these techniques is their strong assumption that the annotated item is textual. This means that the item can be annotated by identifying the relevant words and annotating them with corresponding ontology terms. This methodology does not work for resources that do not explicitly contain their meaning. A good example are multimedia files (video, images) which are frequently present in web-pages.

Automatically detecting the meaning of these resources is a challenging task. Nevertheless there are two main approaches in this direction. The first approach exploits textual sources that are closely related to the item. For example, Google indexes the images by analysing their URL and the HTML code of the page in which they are published. While simple this service of Google has a quite good performance for simple queries. A second approach to this issue is that of emergent semantics [18]. The innovative idea of this research is to derive the meaning of the item by monitoring those interaction of the users with the machine in which the item is involved.

### 2.2 Software on the web

Parallel to the Semantic Web initiative, the web is changing from a collection of passive web-pages to an information space populated by active software components. Many business applications are present on the web in the form of online
services, recently catalogued by the term "web-services". Also, intelligent agents spread on the web. Besides these components with active behaviour, lot of software assets (source code fragments, executable modules, software tools) are published on the web, either gathered in specialised repositories or spread on different web sites.

Some relevant examples of current software-repositories on the web are:

- **UDDI**[^1] (Unified Description, Discovery and Integration) [1] is an industry-wide initiative to support the discovery and the integration of business applications on the web. Businesses can register to UDDI and describe themselves at several levels of abstractions:
  - *white pages* - contains general information about the business, such as address and contact person;
  - *yellow pages* - each business specifies a set of associated services which are described in terms of concepts specified in a product catalogue (UN-SPSC). This information is used at retrieval: the product catalogue is used as the browsing structure;
  - *green pages* - in case a business offers a web-service it can specify the address of a file which contains a functional level description and implementation details of the service. For example one can supply an IDL (Interface Definition Language) file which specifies the interface of a class (the names of the methods and their parameters). Or a WSDL file (see further) which specifies an interface and its implementation details. These technical descriptions are used by the developers who want to integrate a service in their own service.

After using UDDI we conclude that, as it is today, it offers little help both in finding and integrating services.

- **SourceForge**[^2] is the largest repository of Open Source code and applications available on the Internet hosting over 51 000 projects (21.10.2002). This library uses a method which follows the principle of the Faceted Classification technology, introduced in 1985 for software libraries. There are seven facets (orthogonal classification criteria): Development Status, Environment, Intended Audience, License, Operating System, Programming Language, Topic. These facets and their values form a light-weight ontology containing terms related to software components. The nice feature of the browsing mechanism ("Software Map") is that one can narrow a search by imposing a set of filters in different facets. For example the results of a browse action to Topic/Communications/Email/Filters (177 applications) can be filtered to have "Environment = Web". In this case 33 items match the criteria. Further filtering to "Operating System = Unix" yields 14 entries. The same ontology of terms is used as a navigation backbone in the smallest application repository (around 26 000 projects), FreshMeat[^3].

[^1]: http://www.uddi.org/
[^2]: http://sourceforge.net/
[^3]: http://freshmeat.net/
• **Tucows**\(^4\) - is a large application repository offering a broad range of applications from business software to games. The main navigation facility is a broad taxonomy which classifies the applications over several major categories (different operating systems, mobile applications, games). A certain type of application (ex. email clients) exist in different domains (for example in the set of Windows applications as well as for hand-held devices). This light-weight ontology is quite sloppy, often being inconsistent. Some terms are often classified differently: sometimes email clients are considered as related to the "Communication" category, other times you can find them under the "Internet" category. While browsing is quite easy, the search functionality often does not work as expected: searching for "email clients" brings up all sort of clients (ftp clients) and even applications that have nothing to do with email nor are they clients.

• **Download.com**\(^5\) This application repository uses the same method to classify its items as Tucows. However the taxonomy contains different terms. This portal is aimed both for application developers and users. Some parts of the browsing taxonomy are designed for different types of developers, such as web-developers.

• **The Web** Software components exist spread over the whole web, grouped in dedicated repositories as described above or simply published on some web pages. In an ideal case search for a component should not depend on knowing the right repository but could be done over the entire web. However current search engines return all documents related in any way to the terms of a query. Seldom is the case that the returned pages contain a link to a certain software component. Usually it is necessary to take some steps from the returned page in order to find a component.

We can draw the following conclusions:

• There is a lot of software on the web. We have given just a few examples above, but the web contains much more software.

• Finding components is difficult. Search for software is based on key-words matching on annotations (1) added manually or (2) extracted by search engines (Google) from web pages.

• Integration of active components is manual. In case of a set of services that can be integrated in order to support a more complex task (ex. one needs currency converters, ticket bookers, hotel bookers to plan a journey), the integration of these services has to be done manually. No existing description of functionality is machine understandable.

The www consortium recognized these issues with vital implications especially for web-services, and initiated a working group to standardize an architecture

\(^4\)http://www.tucows.com/
\(^5\)http://download.com
for web-services. According to the document emitted by this group[3], the web services architecture has three main parts (called "stacks"), as depicted in Figure 2.2.

The first stack, "Wire", encapsulates the concepts and technologies dealing with the actual physical exchange of information. This includes standards for (1) transporting information (HTTP being the de facto standard, also possible FTP, CORBA), (2) packaging information (XML based massaging protocols such as SOAP), and the possibility to make extensions to the lower layers.

The second stack, "Description", is the stack of description documents defined using XML Schema. There are different aspects of the component which can be described. The Interface specifies the messages that a service expects and returns and the Implementation describes how to encode the messages and where to send them. The Policy specifies additional descriptions such as quality of service, management requirements etc. and the Presentation describes how to render the input/output messages on the screen of different type of devices. Composition and Orchestration define some "work-flows" in executing more services or within a service Business and Service Agreements are not yet specified.

The last stack deals with the items that are necessary for service discovery. No technology is imposed: service publication can be made either to a broker or just locally. Discovery can be done in a centralized or a p2p fashion.

We observe that while the lowest, physical inter-operability stack is already in place (HTTP, SOAP), the second layer is covered only for the specification of interface and implementation (WSDL) and policy (UDDI). The rest of the description layer, namely the work-flow specification and the agreements, are not settled yet. The last layer is also vaguely specified.
We conclude that while the need for describing software components in innovative ways is there, no concrete solutions exist in the industry. Our hypothesis is that augmenting software components with machine understandable semantics would allow automatic support both for search and integration.

2.3 Semantics of software

By semantics of software we mean any description that says something about that component. We chose this broad term because, as we will show in this chapter, a large variety of methods exist to describe a large range of issues about software components.

2.3.1 Traditional approaches

The general idea of describing some aspects of a software component is not new.

Software Libraries are maybe the most relevant example. The Software Reuse community recognised the need for building software libraries for storing and retrieving reusable components. The issue of "finding appropriate components" together with that of "creating reusable components" and "reusing components" was considered of vital importance for a software reuse suite.

Surprisingly, no major reuse technology deals with SL. When comparing three of the main reuse technologies, Forsell [Forsell, 2000] observes that none of them treats the "finding appropriate components" issue. The explanation is that reuse systems are thought of as being active within a company, a small and closed environment. In most cases the number of components is relatively small and it is possible to retrieve the desired component by simply browsing. This is especially easy because the components are homogeneous (in programming language/application domain) and the users have background knowledge.

This is surprising because a lot of methods were proposed for building libraries. An extensive survey [14] overviews these methods and roughly groups them in six groups:

- Information retrieval methods: many researchers view software retrieval as a specialized form of information retrieval. Several efforts report on applying traditional IR techniques to textual documents that describe a component: the source code, a design document or a natural language description. The main challenge for these techniques is to provide good recall and precision. They are applicable for small libraries and give best results if the users have the same interpretation of terminology.

- Descriptive methods: match a keyword-based query against assets that are represented by lists of descriptive keywords. Due to the difficulty of maintaining consistency and accuracy these methods are limited to in-house software libraries of limited size. An outstanding example is the faceted classification approach of Pietro-Diaz [17].
• operational semantics methods: the components are represented by their input/output behavior, i.e. by a set of valid inputs and their corresponding outputs. To query such data one needs to supply a sample input and an oracle, i.e. an agent that can determine if the observed behavior is consistent with the user’s intent. For example, Atkinson [2] use the Z language to specify the input and the corresponding output streams of a component. These methods are usable only for executable components.

• denotational semantics: methods in this group are based on formal specifications which allow to gradually increase the precision of the retrieval. The result is filtered in three steps: signature matching reduces the search space, then model checking and theorem proving improve precision by applying stronger matching criteria. Even if these techniques have a very good precision they have two problems: (1) it is costly to acquire the formal semantics and (2) they are computationally expensive.

• topological methods: are based on the simple premise that, given a query that describes some required features, we are interested to identify assets that come closest to providing these features. There is a large range of research initiatives that define some closeness measure between components. Many AI and logic related solutions were used in this field such as Ostertag’s AIRS (AI based Reuse System) which combines faceted classification and semantic networks [16].

• structural methods: opposite to all previous methods which select components based on their functionality, the methods of this class select a certain component based on its structure. For example two programs that sort an array have the same function but different structure, while two programs one summing up the elements of an array and the other multiplying all elements of the array are different in functionality but very close in structure.

Many successful software reuse programs rely on very simple techniques [14]. Thirteen out of fifteen software libraries surveyed in 2000 [8], used keyword based retrieval and only one used faceted classification. According to [14] this is because software libraries are often not a bottleneck in reuse given the properties of the environment.

**Formal methods** Formal methods use mathematical logic or a specification language with formal semantics(ex. Z) to specify and model the behavior of a system. They were applied also to describe software, not only for retrieval (see denotational semantics) but also to mathematically verify that the system design and implementation satisfy system functional and safety properties. The promise is to discover major flaws in the design phase rather than at test time. A classic formal language for software specification is the formal specification notation Z, useful for describing computer-based systems. It is based on Zermelo-Fraenkel set theory and first order predicate logic.

---

6[http://www.zuse.org/z/](http://www.zuse.org/z/)
The objective of the IBROW project is to develop intelligent brokers that are capable to configure knowledge systems from reusable components on the web. Essential for the project are problem solving methods (PSM) which describe the reasoning part of a knowledge based system. While there exist a large number of specification languages for PSMs, ranging from informal to formal methods, a new specification framework was built in IBROW, UPML [6]. The UPML framework allows describing these components so that they can be semiautomatically reused and adapted.

The idea behind UPML is visionary as it addresses the requirements of the Semantic Web, before the idea of the SW was well-established. UPML uses ontologies to describe the PSMs and it is aimed to be used as a standard for representing knowledge components on the web. Further it is possible to express this framework using SW languages such as RDF(S) [15]. There were even considerations of bridging between UPML and OIL [7].

However UPML is quite restrictive being developed to (1) describe a certain kind of components (PSM) and (2) support a single task: configuration.

Software descriptions on the Web

The idea of semantically describing software components is a natural extension of the SW. If active components would publish their functionality then it would be easier for agents to find and to dynamically integrate these services. The prototype of a system that uses semantic descriptions of web services that are then automatically integrated by agents, is presented in [12].

Providing semantic descriptions of components raises several issues relate to content, ontology and language.

The first big question is: what should be the content of the annotation? While the answer to this question is easy when annotating simple HTML-pages, it becomes non-trivial for special type of items such as multimedia or software. Several aspects of a software component can be described. Usually, the task that is supported by a description determines the content of that description. We will support this statement by examples of existent software descriptions:

- Application repositories base their navigation on a description of the capabilities of the applications (ftp clients, screen savers etc.)
- Code repositories for developers describe their content according to more technical criteria (operating system, development language)
- A WSDL file contains the abstract description of functionality/messages as well as their binding with the implementation details (interface and implementation). There is no formal semantics associated to the names of messages and functions. They have to be interpreted by a human, being simple strings with no meaning for a machine. These descriptions are intended for developers who want to integrate the service in their own tool.

\footnote{http://www.ibrow.org/}
• Web-service repositories (UDDI) supplement the WSDL descriptions with business-level descriptions which are meant to assist humans in their search.

• The DAML-S ontology allows describing several facets of a web-service in a machine processable way which can be used for automatic discovery, integration and monitoring of services.

Second, what ontologies can we use to describe a component? To provide the description of a component one needs to have ontologies that contain domain specific terms. We have already seen that many web-repositories rely on lightweight ontologies, i.e. very simple taxonomies. The DAML-S ontology formally describes web-services [5]. However:

• It only focuses on web services, ignoring passive software components.

• The descriptions aim to facilitate automatization of a large scale of tasks: from finding the services to integrating them. In our opinion, even the first task is very ambitious.

• DAML-S assumes high quality pre/post conditions. These can be only built by hand.

Third, what semantic language should be used to express the ontologies and the annotations? DAML-S is layered on DAML, however it is not sure that general purpose semantic web languages will suffice to describe software components.

While research currently focuses on how to describe components, we envision that the provision of semantics will be soon recognised as a problem. Pioneer work in this direction was done by E. Mena [13]. He proposes scraping several online repositories (Tucows.com and Download.com) and merging the so-obtained ontologies in a global ontology (SoftOnt) using WordNet. Already this light-weight ontology enhances the search for components.

2.3.3 Comparison

After the previous two sections one might wonder: why do we need new methods to describe software components on the web? Is it not enough to use the methods developed for traditional software libraries?

Some traditional methods are already used on the web.

• information retrieval methods are used by search engines to index web-pages that are related to software components. They are also the bases of the search functionality in application repositories (ex: Tucows). In this case the textual corpus that is used is composed of the natural language descriptions of the applications provided by the authors. These methods provide a good recall but a very poor precision. While this limitation was not so problematic in small repositories, it makes the method unusable for large collections of software components such as the web.
• descriptive methods are used to organize and to browse application repositories (ex. SourceForge). Such methods are usable as long as the taxonomy describing the components is not too large. They were successfully applied in repositories where the applications are homogeneous from a certain point of view. For example the SourceForge taxonomy describes the main characteristics that are interesting for open source software. These methods fail for collections of heterogeneous components which cannot be classified according to certain facets.

These methods fail for large repositories with heterogeneous components bridging over several domains. The idea of the Semantic Web addresses this issue of describing heterogeneous knowledge with the help of ontologies.

Formal methods were not applied on the web. However existing semantic web languages are grounded in logic which provides a machine understandable semantics. While traditional formal methods are computationally expensive leading to large execution times, current semantic languages (OWL) strive to be formally less heavy.

One major problem perpetuates from the traditional techniques to novel techniques: the acquisition of software descriptions (semantics). Each method uses some description about the components ranging from light-weight ontologies to formal specifications. The cost of acquiring these descriptions manually increases from simple IR methods to formal methods. It is considered a reason why most reuse systems are based on the simple/cheap methods. This issue is valid for the novel, Semantic Web techniques too: both for ontology building and producing annotations for the components.

3 Research statement

The main goal is to ease ((semi) - automate) the acquisition of software semantics. We will do this by applying different (AI) techniques to existing resources about software (UML), with a high emphasis on resources available on the web (mailing lists, libraries, UDDI). We verify our extracted semantics by employing them in software search on the web, however, the results of our work should be applicable in any general setting (for example, searching software assets in intranets).

We can summarise our research interests in two questions:
(1) Is it possible to semi - automatically extract software semantics from existing (web) sources?
(2) Does the quality of these semantics suffice to enhance search for software components on the web?

By semantics we mean:
(1) ontologies - which formally describe the software domain
(2) annotations - the descriptions of software components in terms of the concepts defined in ontologies.
In relation to the first question, there are two main issues to investigate:

1. Determine resources that contain information about software components. These resources range from by-products of the software engineering process (UML diagrams) to web resources (web-pages, mailing lists or repositories such as UDDI and Sourceforge).

2. Design methods for extracting semantics from the resources. The methods for semantics extraction vary for different sources.

We can test the quality of the extracted semantics by using them to support a certain task. For a start, we will use the extracted semantics to facilitate software component retrieval on the web. However, other applications are also possible.

4 Concrete plans

We have significantly restricted our focus within the Semantic Web research, so that we could define the interest of our work. To go further we need to take a closer look to several issues.

4.1 Envisioned research strategy

There are basically two approaches to work on the problem:

- A top-down approach in which the global picture is detailed to great length resulting in a methodology on how to extract and to use semantics. The correctness of this methodology would be verified by applying it in several case studies. The difficulty of this approach consists in working without actual knowledge of what happens in reality.

- A bottom-up approach in which more case studies are carried out and then finally a methodology emerges.

We will combine these approaches in an iterative fashion, i.e. alternating between case-studies and building a high-level framework. We start out with a case study which will provide enough experience to sketch the framework. The next case-studies will verify and enrich this framework.

As a first case study we have chosen to enhance search for software components on the web. Because we know little about the domain of searching for software on the web and we do not have a clear picture about the existing problem(s), we will follow these steps:

- Assess the existing technology. As a first step we will identify the problems related to search for software components. This implies some tests in which we look for different types of software components (web-services, code samples), from different points of view (functional, operational). This would already give us an insight of what content can be described about components, what kind of components exist and also about users’ practices. And most important we can assess the performance of existing technology and identify its short-comings. Shortly, we hope to answer the following questions:
– what components are on the web?
– what do users’ look for?
– how do users’ search for software?
– to what extent can users find what they are looking for?
– which are the shortcomings of today’s technology?

The scenario slightly changes if software agents search for components. We expect changes relating to the way they will build queries and their expectations for the results of the search. Maybe they will use the same queries as human agents (they just translate the needs of their user). However, unlike human users, they will expect as answer a direct pointer to the component (URL), rather then to an artefact related to the component.

• Apply Semantic Web technology to enhance existing technology. Once the problems of existing technology are clear, we will try to solve them using SW technology.

– Chose an infrastructure - decide on the architecture of the search engine, i.e. how will it be populated, what storage/reasoning engine it will use? We could simply adapt ideas from existing initiatives [13], [12]. At this point we also need to decide on the semantic language used to describe ontologies/annotations.

– Build ontologies - our claim is that the result of search engines can be enhanced if they posses existing domain knowledge, modelled in ontologies. Therefore we will build a set of domain related ontologies (which would specify for example the subclasses of sorting algorithms and that the speed of the sorting algorithm is the main efficiency measure). For this purpose we can design some automatic methods. For example, scraping light-weight ontologies from existing web resources (online catalogues, Google), defining domain specific ontologies.

• Extending the Semantic Web solutions In the previous point we plan to acquire some ontologies from scraping data that is on the web. Further we will gradually enrich these ontologies and the used semantic descriptions by mining other software related sources too, such as UML.

4.2 Open Issues

Figure 4.2 is an "instantiated" version of 2.1 showing the main concepts involved in our work. Opposite to the general figure 2.1, we will not deal with any web-resource but we will focus our research on software. The descriptions associated to a component will support a certain task: software retrieval on the web. Our main interest is in automatically building some descriptions. Our view is that both content and ontologies can be extracted from certain sources, as we show on the figure. There are several open issues related to these concepts that we need to consider.
The retrieval task When building a search engine for software components one needs to decide on:

- the type of retrieved components - as we described in section 2 there are several types of components on the web, ranging from code samples to web-services and agents. It is important to decide whether we will support the retrieval of a certain or general type of component. To do this we need to know: what components are there on the web?

- the audience (implying the kind of questions they will ask): computer users usually look for applications with a certain functionality or web-services that provide a certain service. We expect that developers would look for certain components that perform a programming function (search lists, concatenate strings). This choice determines the content of a given description. What are people looking for?

Content From the examples of software descriptions we have seen that there are several ways to describe a software component ranging from capability descriptions (application repositories) to describe functionality and implementation details. We could also draw the conclusion that the type of audience (developer or user), the type of software components (active or static components) and the nature of the task (retrieval or integration) all influence the content of the description.

There are several resources that contain information about software components. These resources can be:

- by-products of the software engineering process (all flavours of UML diagrams, from Use Cases to Class Diagrams)

- web - resources: web-pages, mailing lists or special repositories such as UDDI and Sourceforge.
Special services on the web: for example Google-sets is an excellent tool to support building ontologies for software description. Suppose we want to extend our domain description with all existent programming languages. It is enough to know 2-3 such languages, Google set will return the rest. In the same way we can exploit information gathered in web-directories (ODP), just by focusing on the categories related to software.

Methods for extracting semantics from the resources. The methods for semantics extraction vary for different sources. For example translating UML diagrams into DAML implies defining a mapping between the elements of UML and DAML and writing some translation software applying these mappings (XSLT/java etc.). For less-structured resources (e.g. mailing lists, web-pages) concept extraction software, data mining techniques or text-scrapers are needed.

4.3 Planning

An estimative planning for the upcoming three years follows.

- First year:

  - 1-6 months: Acquiring knowledge about software, software descriptions, search, current search engines. Approximately 2 months per each activity.

    * Generic UML to DAML(-S, -L) transformer. The mission of one of the master students is to develop a tool for transforming an XMI serialisation of an UML diagram to DAML, using different mappings: the XMI and an XML file specifying the mapping are the input, the DAML representation the output. The student will focus on class diagrams, however UML contains dynamic diagrams and OCL constraints as well. We should build the tool so that it is possible to translate activity diagrams to DAML-S (DUET comes with some ideas for the mapping\(^8\)). Another issue is how to translate OCL rules. There is some rumour about the DAML-L rule language, but I cannot find anything related to it.

    * SW@VU - describing simple web-services. In our SW@Vu project we rely on using quite some web-services: a bib2rdf translator, an rdf-validator, Sesame, sameIndividualAs identifier. Currently we integrate and use these services manually. It would be much better if we could write a program that calls these individual services to achieve a complex goal. We will describe the functionality of the services in a semantic language (DAML-S) and use these descriptions to integrate them automatically. This is a small exercise to get a feeling about active services and DAML-S.

    * Why is Google not enough? Some people claim that Google is just enough for finding any software on the web. If this is the case we

\(^8\)http://codip.grci.com/codipsite/wwwlibrary/DUETGuide/DAMLs-UML_Mapping_V1.htm
do not really have a problem. We will perform some case studies of searching the web for different kind of components, by different types of users. We will get insight in what components exist, what aspects are relevant for users and also where does Google fail.

- 6-12 months: Applying semantic web technology to enhance the problems that we have detected before. Build a first prototype of a search engine based on existing sw technology.

- Second year:
  - 1-6 months: If it is the case, enrich the semantics of the search engine with other type of descriptions generated from software related sources (ex. UML)
  - 1-3 months: Build a framework about generating and using software semantics, based on the lessons learned in the first case-study.
  - 1-3 months: Carry out another case-study to verify the framework.

- Third year:
  - 1-9 months More case studies + changes in the framework.
  - 1-3 months - write thesis

References


