SEMANTIC APPLICATION DESIGN

Philippe Larvet
Alcatel-Lucent Bell Labs, Centre de Villarceaux, 91620 Nozay, France

Keywords: Application design, software component, semantic component.

Abstract: This paper presents a process to determine the design of an application by building and optimizing the network of semantic software components that compose the application. An application has to implement a given specification. We consider this specification is made of atomic requirements, logically linked together. Each requirement is expressed in natural language: this expression is seen as the semantic description of the requirement. Off-the-shelf components from which we want to build the application can also be described through a semantic description. We consider a component implements a requirement if the "semantic distance" between their two semantic descriptions is minimal. Consequently, designing an application consists of building and optimizing the logical network of all semantic optimal couples "requirement-component". The paper presents such a building and optimization automatic process, whose development and improvement are still in progress, and whose main advantage is to systematically derive the discovery and assembly of software components from the written specification of the application.

1 PROBLEM OF APPLICATION DESIGN

Software application design is traditionally a complex activity. According to the accepted definitions used as references in the scope of object-oriented and component-based application development, and according to Grady Booch (Booch, 2007), design is "*that stage of a system that describes how the system will be implemented, at a logical level above actual code. For design, strategic and tactical decisions are made to meet the required functional and quality requirements of the system. The results of this stage are represented by design-level models: static view, state machine view, and interaction view."* The activity of design leads to the architecture of the application, which is "*the organizational structure of a system, including its decomposition into components, their connectivity, interaction mechanisms, and the guiding principles that inform the design of the system."* (Rumbaugh, Booch, Jacobson, 1999).

Many authors have described several methods to guide the building of component-based applications (Bordeleau, 2005; Chusho, 2000; Kirtland, 1998) but except within the field of semantic web services (Narayanan, 2002), it seems an automatic semantic-oriented process has not been considered as a serious approach to design. Traditional component-based development approaches have two main drawbacks: they are often fully manual, and the process of finding and assembling the right components is not directly derived from the text of the written specification.

Notice that research in semantic web services (Narayanan, 2002; Patel-Schneider, 2002) proposes a semantic-oriented design approach, by using a *logic-based* approach to the specification of semantics, whereas the approach presented in this paper uses *natural language* as the basis for specifying semantics.

An application has to rely on a given specification. We consider this specification exists under the form of a natural language, informal text, that describes functional and non functional requirements the application has to cover. We have made three main assumptions in this paper:

- a software application can be built by assembling off-the-shelf components;
- the determination of components can be derived from the semantic analysis of the requirements;
- application design, i.e. architecture of solution, can be derived from the architecture of the problem, i.e. from relationships between requirements.

Larvet P. (2008).
SEMANTIC APPLICATION DESIGN.
In *Proceedings of the Third International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 47-55
DOI: 10.5220/0001761900470055

2 THE PROPOSED PROCESS

We see an application as a set of inter-related components. Each component has a functionality, expressed as a set of functions, and encapsulates and manages its own data: this is the component paradigm, derived from object-orientation and today's standard of development.

Let us consider we have at our disposal many small off-the-shelf components, stored in appropriate component repositories. Each one covers a precise elementary function, an atom of functionality – for example file management, database access, GUI display mechanisms, text translation, HTML pages reading from URLs, elementary functions for text processing, etc. More simply than the concept of semantic component (Kaiya, 2005; Sjachyn, 2006; Hai, 2006), we propose each component is described through a *semantic card* which contains notably the *goal* of the component, expressed in natural language form and describing clearly what the component really does, what its functions are and which data it manipulates.

Through an appropriate process we expose in detail below, the *meaning* of the sentence representing the component's goal - its *semantics* – can be determine and expressed in terms of an appropriate computable data structure. Thus, the idea is to mark every *semantic atom* of functionality with their appropriate semantic data structure.

We also have at our disposal a specification document containing requirements describing what the application will do, what its functional and non-functional features are. The requirements are a set of sentences expressed in natural language. Each sentence has a meaning which can be found out by using the same process. Each sentence, i.e. each piece of specification, each atom of requirement, can therefore be evaluated and marked, and each sentence will receive its own semantic data.

Notice that this process is different than an ontology-based requirement analysis approach (Kaiya, 2005) – an ontology (McGuinness, 2004) is a formal description of the concepts manipulated in a given domain and of relationships between these concepts. Here, no external ontology is used to help requirements analysis, because semantics is extracted from the text itself.

Sentences that compose requirements are logically linked to each other. Then, it is possible to determine a *requirement network* by scanning links between requirement atoms: this browsing will determine the structure of the 'specification molecule' – the molecule that describes the problem.

Analyzing lots of specifications within the context of numerous industrial projects developed with an object-oriented approach (Larvet, 1994) has led us to observe that a link between two different requirements in the specification always leads to a link between the classes implementing these requirements. Indeed, two pieces of requirement are linked to each other when they both talk about a given data, constraint, functionality or feature of the targeted application. Then, the same link exists between the components implementing these requirements.

Consequently, it makes sense to consider that links between the bricks of the problem have a similar correspondence to links between the blocks of the solution. In other terms, problem structure – 'specification molecule' – is isomorphic to solution structure – 'design molecule'.

Our proposed process consists of three steps:

1. finding the components whose semantic distance is the shortest with semantic atoms of requirements;
2. organizing these components in order to constitute the 'solution molecule', i.e. the initial architecture of the application – this initial design being made by replicating the problem molecule and using solution atoms instead of problem atoms – but these kinds of atoms do not have exactly the same nature, so the initial component interaction model has to be optimized; and
3. optimizing the structure of solution molecule in order to determine the best component interaction model.

Within this approach, the initial component interaction model – corresponding to the initial design of the future application - is built from relationships between application's requirements: an association between two requirements will determine an association between the two components that cover these requirements.

3 SEMANTIC CARDS FOR COMPONENTS

Semantic cards (*semCards)* formally describe the small off-the-shelf components that are used to build applications. Each semCard contains the goal of the component and the list of its public functions with their input and output data. We propose a semCard has an XML representation where input and output data are described with three main attributes:

1. a data *name*;
2. a *concept* associated with the data, expressed in reference to a word defined in an external dictionary or thesaurus, in order to specify the semantics of the data; here, the concept belongs to a *domain* addressed by the component and whose name is mentioned in semCard's header; and
3. a *semantic tag*, or semTag, of the data, which represents a stereotype of a semantic data type and specifies the nature of the data (Larvet, 2006); this semTag will be useful to determine and optimize components' interactions.

The semantics of the operations' goals is defined with precise rules that help to write terse and non ambiguous expressions:

- goals are expressed in natural language, using specific words;
- these words belong to 'lists of concepts' that are embedded in the semCard and summarize the pertinent words to be used to write goals; and
- words composing 'lists of concepts' are defined in external dictionaries and belong to related domains that are referenced in the semCard.

Ontologies could be used to summarize and formalize the definitions of concepts and domains, but this is not mandatory. RDF (Patel-Schneider & Siméon, 2002) or OWL (McGuinness, 2004) are convenient to depict such ontologies, because they are standard and well-tooled languages, but simple ad-hoc appropriate XML files containing word definitions and domain descriptions are also suitable.

Here is, as an example, the semCard for an RSS-feed-accessor component:

```
<semCard>
<URL>http://xxx.xx.xxx.x/components/RSS
/RSS_Component.asmx</URL>
  <component name="RSS">
    <domains>
      <domain name="RSS">
        <concepts list="RSS, RSS_feed,
URL, news" /></domain>
      <domain name="News">
        <concepts list="news, title,
titles, description, article, text,
News Agency" /></domain>
    </domains>
    <operation name="getAllTitles">
      <goal>The goal of the operation
getAllTitles is to deliver the titles
of all the news of the RSS feed
addressed by a given URL.</goal>
      <input name="URL_RSS"
concept="RSS#URL" semTag="URL" />
      <output name="titles"
concept="News#title" semTag="text" />
    </operation>
    <operation
name="getDescriptionOfTitle">
      <goal>The goal of the operation
getDescriptionOfTitle is to deliver the
description of the given title of one
news of the RSS feed addressed by a
given URL.</goal>
      <input name="URL_RSS"
concept="RSS#URL" semTag="URL" />
      <input name="title"
concept="News#title"
semTag="short_text" />
      <output name="description_
of_title" concept="News#description"
semTag="text" />
    </operation>
  </component>
</semCard>
```

4 DETERMINING THE MEANING OF SENTENCES

One key to our process is the possibility to compare the meaning of a requirement extracted from the specification document with a component's goal, written in the component's semCard. This comparison is done in order to be able to choose this component because it is intended to cover the requirement.

The key to this comparison is the ability to determine the *meaning* of a text. We consider this meaning is made up of the concatenation of elementary meanings of all the pertinent terms that compose the text. The ability to compare the meaning of two different texts implies the ability to compare two different terms and to determine whether they are semantically close or not.

Important works have been done related to *semantic proximity* in natural language expressions (Khaitan, 2006; Corley, 2005; Guha et al. 2003; Mayfield and Finin, 2003; Guarino et al., 1999; Evans and Zhai, 1996). The novelty of our approach is to propose a way to express the meaning of an elementary term in order to process a comparison with another term. To do so, we build a "synonym vector" with the synonyms of the term that can be found in a thesaurus, and we call it a *synVector*.

For example, the synVector of "battle" is:

```
battle = {fight, clash, combat,
encounter, skirmish, scuffle, mêlée,
conflict, confrontation, fracas, fray,
```

action; struggle, crusade, war, campaign, drive, wrangle, engagement}

Other examples:

war = {conflict, combat, warfare, fighting, confrontation, hostilities, battle; campaign, struggle, crusade; competition, rivalry, feud}

peace = {concord, peacetime, amity, harmony, armistice, reconciliation, ceasefire, accord, goodwill; agreement, pact, pacification, neutrality, negotiation}

Table 1: Functions defined on synVectors.

Notation	Semantics
synV(word)	synVector for the term 'word'
card(V1)	cardinal(vector V1)
common(V1, V2)	{syn_1, syn_2, …, syn_n} | $syn_i \in$ V1 and $syn_i \in$ V2 Example: `card(common(synV("battle"), synV("war"))) = 9`
avg(V1, V2)	average(cardinals(V1,V2))
semProx(T1, T2)	`100 * card(common(synV(T1), synV(T2))) / avg(synV(T1), synV(T2))`
phraseVector(sentence)	{synV(Ti)} | Ti ∈ {T1, T2, … Tn} and Ti = pertinent word of <sentence>
semVector(sentence1, sentence2)	{best values of comparisons among all couples synV(T1),synV(T2) | T1 ∈ sentence1 and T2 ∈ sentence2}
diff(semV(req1,req2), semV(req1,req3))	abs(semVector(req1,req2) – semVector(req1, req3))

The concept of *semantic proximity* between two terms T1 and T2 = semProx(T1,T2), gives a ratio taking into account common synonyms within the two synVectors of T1 and T2. If semProx is greater than a given value A (for instance 50) or close to 100, we consider the two terms are semantically close. For example, `semProx("battle", "war") = 100 * 9 / 0.5 * (19 + 13) = 56.25`. In other words, in the union of synonyms sets for "battle" and "war", 56% of the elements are found in duplicate. Inversely, if semProx is less than a given value B (for instance 10) or close to zero, the two terms are semantically distant. Obviously, for instance, `semProx("war", "peace") = 0`.

Values of levels A and B can be "tuned", according to the category of texts to be processed.

The determination of the meaning of a given sentence is made as follows:

- the sentence is analyzed and pertinent words are extracted – non-pertinent words like articles, prepositions, conjunctions, etc, are ignored;
- for each pertinent word, a synVector is built;
- a vector of vectors for the whole sentence - a *phraseVector* - is built by assembling all synVectors of pertinent words contained in the sentence, as shown in Figure 1.

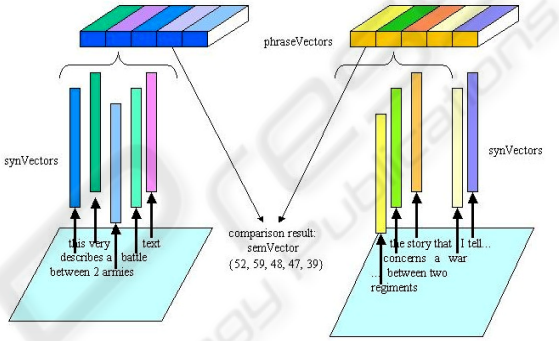

Figure 1: Building a semVector from phraseVectors of two sentences.

For example, let us build a phraseVector for the following requirement, extracted from the specification of a Call Management System:

requirement = The caller makes a call to a receiver by creating a message that contains the call subject, submitted to the receiver at the same time

Pertinent terms are: caller, call, make a call, receiver, message, subject, submit.

The phraseVector for this requirement is the concatenation of the following synVectors:

synV(caller) = {phone caller, telephone caller, visitor, guest, company}(5)

synV(call) = {phone call, telephone call, buzz, bell, ring; demand, request, plea, appeal, bid, invitation}(11)

synV(make a call) = {phone, make a demand, send a request}(3)

synV(receiver) = {recipient, heir, addressee, beneficiary, inheritor, heritor}(6)

synV(message) = {communication, memo, memorandum, note, letter, missive, dispatch}(7)

```
    synV(subject) = {topic, theme,
focus, subject matter, area under
discussion, question, issue, matter,
business, substance, text; field,
study, discipline, area}(15)
    synV(submit) = {offer, present,
propose, suggest, tender}(5)
    phraseVector(requirement) =
{synV(caller), synV(call), synV(make a
call), synV(receiver), synV(message),
synV(subject), synV(submit))
```

The comparison of three sentences S1, S2 and S3 is made by comparing their phraseVectors (see Figure 1). This comparison builds a result that will be used to calculate the *semantic distance* between the sentences. Let us detail the phraseVectors comparison steps:

- internal synVectors of the two phraseVectors are compared two by two – this means every synVector in S1 is compared to every one in S2 and S3;
- a semantic proximity (semProx) is calculated for each pair;
- the best values of semProx among all comparisons are kept in an ordered external *semantic vector*, a *semVector*, as a result of the comparison; then
- the comparison of semVectors for sentences S1 and S2, and for S1and S3, allows to determine whether S1 is semantically closer to S2 or S3.

The search of components that cover a given specification follows the *phraseVector* approach:

- phraseVectors of requirements are built;
- phraseVectors of components' goals are built;
- phraseVectors are compared and the corresponding semVectors are built for every pair requirement-component; and
- the best semVectors are kept and help to determine the components that are able to fulfill the requirements.

5 DETERMINING THE PROBLEM NETWORK

The *requirement network* summarizes and represents the links between the requirements.

The *phraseVector* approach reveals the links between the requirement atoms and helps the building of 'problem molecule': the sentences of the specification are semantically compared two by two, phraseVectors are built and semVectors are calculated.

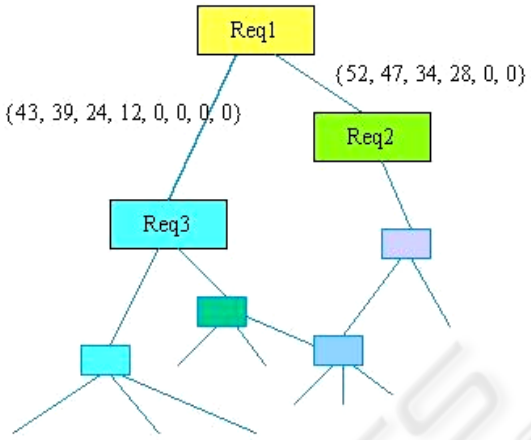

Figure 2: Building the requirement network.

The result, for each requirement, is a set of vectors that represent the links, in terms of semantic distance, of each requirement with respect to the others. We can "tune" the level of this semantic distance to keep only the "best" semVectors in terms of semantic proximity, i.e. the most semantically pertinent links for a given requirement. This means each requirement has a limited number of semantically closest other requirements; in other terms, a requirement can be formally described by a limited set of semVectors that represent the semantically closest other requirements.

The links can be represented in a 2D or 3D space; the aim is to get a convenient model of the problem, i.e. a representation we can communicate and we can structurally compare to another. On this model, we make graphically appear the links between requirements. Only the best links are kept, i.e. the links whose semVector value is larger.

For example, Req2 is linked with Req3 and Req5, but

```
semVector(Req2,Req5)>semVector(Req2,Req3)
```

this means the semVector resulting of the comparison between Req2 and Req5 is greater than the semVector resulting of the comparison between Req2 and Req3, then only the link Req2-Req5 will be kept on the final model. This is a question of optimization. Tuning the model is possible by determining the maximum acceptable gap between two semVectors.

6 BUILDING A PRIMARY SOLUTION NETWORK

We assume that the structure of the solution, i.e. the architecture of the design, is isomorphic to the

structure of the problem. Solution molecule has the same spatial structure as problem molecule, although they do not contain and use the same kinds of atoms: problem atoms are requirements, solution atoms are components. Problem atoms are linked together because they share the same concepts and address the same requirements, solution atoms are linked together because they share or exchange the same data, however the network that links the requirements together contains the same paths as the network of the solution.

The problem now consists of finding the components whose semantic distance is the shortest from the semantic atoms of requirements, and organizing these components in order to constitute the solution molecule, i.e. the architecture of the application that will suitably solve the problem expressed in the specification document.

To build this organization, we will apply the following steps:

- find the components that cover the requirements by using the semVector approach; this will build a list of components, not yet linked together (see Figure 3);
- replicate the structure of the problem molecule inside the components list using solution atoms instead of problem atoms, i.e. by attaching to the corresponding components the links between the requirements they fulfill; this will build a rough version of the solution molecule;
- and finally optimize this primary version in order to determine the best structure for the solution molecule. This will become the final architecture for the application.

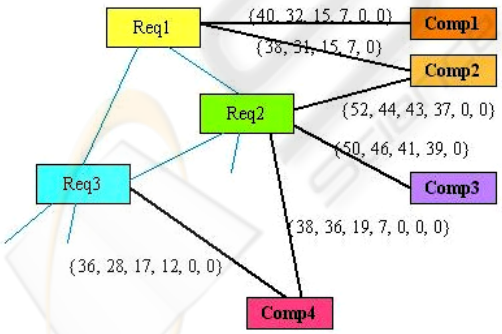

Figure 3: SemVectors help to determine which components fulfill which requirements.

The optimization process will use semantic tags attached to data descriptions of components' operations to determine and optimize interactions between components. The final result of this process is an interaction diagram showing coupling and interdependencies between components.

Replicating requirements links inside components' structure associates components in the same way requirements are associated in the specification; but of course these associations are not all valid: the fact that two requirements share the same concepts does not necessarily imply the two corresponding components have an interaction.

The role of the optimization process is to keep only the most useful of the links inherited from the problem molecule, i.e. the associations corresponding to actual data exchanges between components.

7 OPTIMIZING THE SOLUTION NETWORK

In order to automatically determine real connections corresponding to actual data exchanges between components, we use the *semantic tags* (*semTags)* added as semantic metadata to inputs and output of components' operations (Larvet, 2006).

If these *semTags* are suitably chosen and set, components can be connected and their connectivity can be formally expressed.

For example, if output of Comp1.operationA semantically fits with input of Comp2.operationB, then Comp1 can be connected to Comp2 through the link "output of A" to "input of B".

So, we can write:

```
out_A=Comp1.operationA(parameters);
out_B=Comp2.operationB(out_A);
```

or, more directly:

```
out_B=Comp2.operationB(Comp1.operationA
(parameters));
```

This means the two connected data have the same semantic "dimension", i.e. they are process-compatible; they share not only the same data type, but the same nature of data. *SemTags* express semantic data types and are similar to UML tagged values (Rumbaugh et al., 1999); they are attached to inputs and outputs within the semCards, and ensure the consistency of components' interfaces; for this reason they are important elements for optimizing components interactions (see for instance semtags in RSS-feed-accessor semCard, in paragraph 3.)

7.1 Automating the Optimization of Solution Network

An example will help us to describe the process that takes into account semantic tags in order to build an

automatic assembly of components. Suppose we want to produce a translated version of a news feed. This requirement is expressed in natural language in the specification, and the semVector-plus-component-discovery approach has allocated two components to this requirement: a RSS-accessor component and a Translator component.

The RSS component aims at gathering information from RSS feeds accessible via Internet, and its interface contains two operations: `getAllTitles()` gets all the main titles of the feed for a given URL, and `getDescriptionOfTitle()` gets the text of the short article for this title.

The Translator component is a classical one whose operation `translate()` transforms a text (given as an input parameter) written in a given source language (input parameter) into a translated text (output) written in a destination language (input parameter).

The problem is to assemble automatically and logically these two components, i.e. their three operations (see Figure 4) in order to fulfill the original requirement: *provide a translated version of a news feed*.

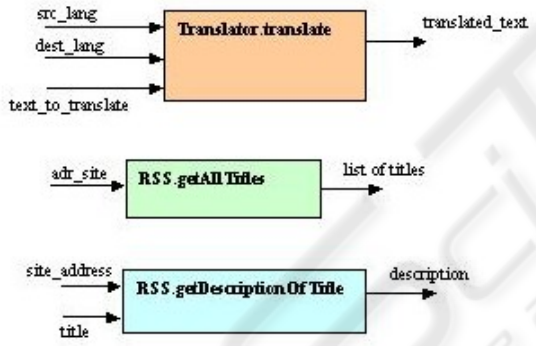

Figure 4: How to assemble these 3 operations?

The first key is to consider semantic tags as inputs and outputs of operations, instead of data. Then, some possible connectivities appear (see Figure 5), but not precisely enough to make a fully consistent composition.

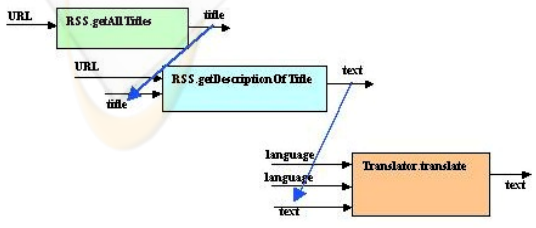

Figure 5: Possible connections (in blue) appear by considering semantic tags instead of data names.

The second key is to consider the *main output* of the targeted component assembly in order to find which operations can provide its inputs, and to iterate the process for these operations: search which other operations can provide their inputs. Then, we go back progressively from the main output to the input data necessary to produce it, and in doing this, we automatically assemble the different operations by linking their outputs and inputs.

At the same time, the links are stored in a FILO (first in, last out) stack under the form of pseudo-code expressing the operation calls. At the end of this process, the content of the stack represents the correct interactions between the components.

The main output of the component assembly is given by the expression of the original requirement. For our example, a *translated version* is wished: the main output is a translated text, i.e. the output of the operation `Translator.translate()`. We can push this main output in the stack, expressed as the "return" of the function represented by the targeted component assembly:

```
translated_text =
Translator.translate(text_to_trans
late, src_lang, dest_lang);
return translated_text;
```

Let us go back now to the inputs of this operation, whose semantic tags are "language", "language" and "text". A data with a semantic tag "text" is provided by the operation `RSS.getDescriptionOfTitle()`.

Then, we can connect this operation to `Translator.translate()`.We can add the call to the operation `RSS.getDescriptionOf-Title()` in the stack, linking with `Translator.translate()` through the name of the exchanged parameter:

```
text_to_translate =
RSS.getDescriptionOfTitle(site_add
ress, title);
translated_text =
Translator.translate(text_to_trans
late, src_lang, dest_lang);
return translated_text;
```

Now, let us go back to the inputs of `RSS.getDescriptionOfTitle()`, whose semantic tags are "URL" and "title". A data with a semantic tag "title" is provided by the operation `RSS.getAllTitles()`.

So, we can also connect these two operations by pushing a new operation call in the stack:

```
titles =
RSS.getRSSTitles(adr_site);
text_to_translate =
RSS.getDescriptionOfTitle(site_add
ress, title);
```

```
translated_text =
Translator.translate(text_to_trans
late, src_lang, dest_lang);
return translated_text;
```

With all the components allocated to the original requirement being used and connected together, the stack now contains the general texture of the component assembly, under the form of a nearly executable pseudo-code. However, this pseudo-code must be refined before it can be executed:

- the data types must be taken into account; for example, `RSS.getAllTitles()` returns an array of Strings and not a single String;
- the names of some parameters can be solved through their semantics, i.e. with the help of their semTags: for instance, "adr_site" and "site_address" recover the same concept and have the same semTag;
- some other parameters can be solved with some useful information contained in the original requirement; for example, if the requirement specifies a *french* translation, then the parameter "dest_lang" of the operation `Translator.translate()` has to be set to "french"; and
- some additional components or operations can be used to solve other parameters; for example, the parameter "src_lang" can be set by using a utility component, a "Language Finder", to automatically determine the source language of a given text, or an operation `getSourceLanguage()` on the RSS feed component.

A specific module, whose detailed description is outside the scope of this paper, makes these refinements in order to complete the pseudo-code:

```
Vector ComponentAssembly(String
site_address) {
   Vector result;
   titles =
   RSS.getAllTitles(site_address);
   foreach title in titles {
      text_to_translate =
   RSS.getDescriptionOfTitle(site_add
   ress, title);
     source_lang =
   LanguageFinder.getLanguage(text_to
   _translate);
     translated_text =
     Translator.Translate(text_to_tra
     nslate, source_lang, "french");
      result.add(title + translated_
                          text);
     }
     return result;
}
```

This pseudo-code can finally be transformed into an executable Java file for example, in order to test the validity of the component assembly produced by the optimization process.

The final interaction diagram between the components, obtained as a result of the optimization process, can be considered as a first draft of the design of the future application. The interest of this draft is to be delivered with a quasi-executable pseudo-code allowing validation tests of the architecture of the future application.

8 CONCLUSIONS

The paper has described an application of a natural language (NL) technology combined with a component-composition optimization process in order to allow the automatic construction of software applications. We have presented an original but partly operational process to determine the meaning of a NL text, and to use this meaning to find the right components fulfilling original NL-expressed requirements of an application specification. This process leads to an initial architectural structure of the targeted application, optimizable with a complementary process in order to get an acceptable and testable draft of the application design.

Among some advantages of this approach, notice that it is performed rapidly and fully automatically, it works directly from the original application requirements and delivers a quasi-executable pseudo-code as a useful sub-product allowing a validation of the future application's architecture. Moreover, traceability between requirements and architecture is guaranteed.

Still in progress, the process has to be improved and refined. An important part of the future work is to do more complete and rigorous experimentation, validation, and perhaps tuning.

REFERENCES

Booch G., 2007. "Object-Oriented Analysis and Design with Applications", 3rd Edition – Cased, Addison-Wesley (2007), ISBN 9780201895513

Bordeleau F., Hermeling M., 2005. "Model-Driven Development for Component-Based Application Portability", COTS Journal, August 2005

Chusho T., Ishigure I., Konda N., Iwata T., 2000. "Component-based application development on architecture of a model, UI and components," *apsec*, p.

349, Seventh Asia-Pacific Software Engineering Conference (APSEC'00).

Corley C. and Mihalcea R., 2005. "Measuring the Semantic Similarity of Texts". Proceedings of the ACL Workshop on Empirical Modeling of Semantic Equivalence and Entailment, page 1318, Ann Arbor.

Evans D. and Zhai C., 1996. "Nounphrase analysis in unrestricted text for information Retrieval", Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics, 1996.

Guarino N., Masolo C., Vetere G., 1999. "OntoSeek: Content-Based Access to the Web", IEEE Intelligent Systems, Vol. 14, No. 3, May/June 1999.

Guha R., McCool R., Miller E., 2003. "Semantic Search", Proceedings of 12th international conference on World Wide Web, Budapest, Hungary, May 2003.

Hai Zh., 2006. "Semantic component networking: Toward the synergy of static reuse and dynamic clustering of resources in the knowledge grid", Oct. 2006, Journal of Systems and Software, V79, 10, p.1469-82.

Kaiya H., Cai Saeki, Ohnishi A., 2005. "Ontology-based requirements analysis: lightweight semantic processing approach", Sept. 2005, Proceedings. Fifth International Conference on Quality Software (QSIC 2005), p.478

Khaitan S. et al., 2006. "Exploiting Semantic Proximity for Information Retrieval", available at http:// www.cse.iitb.ac.in/~pb/papers/IJCAI-CLIA-Exploiting-Semantics.pdf

Kirtland Mary, 1998. "Designing Component-based Applications", Microsoft Press; Pap/Cdr edition, December 1998, ISBN 978-0735605237

Larvet Ph., 1994. "Analyse des systèmes, de l'approche fonctionnelle à l'approche objet", InterEditions, Paris.

Larvet Ph., 2006. "Composing Automatically Web Services through Semantic Tags", ICSSEA 2006, International Conference on Software and Systems Engineering and their Applications, CNAM Paris (France), December 2006.

Mayfield J. and Finin T., 2003. "Information retrieval on the semantic web: Integrating inference and retrieval", Proceedings SIGIR 2003 Semantic Web Workshop.

McGuinness D.L., van Harmelen F., 2004. "OWL Web Ontology Language", W3C Recommendation 10 February 2004, Editors: Knowledge Systems Laboratory, Stanford University, Vrije Universiteit, Amsterdam

Narayanan S., McIlraith S., 2002. "Simulation, Verification and Automated Composition of Web Services", Proceedings of the Eleventh International World Wide Web Conference (WWW-11), pp. 77-88, May 7-11, 2002, Honolulu, Hawaii, USA.

Patel-Schneider P., Siméon J., 2002. "The Yin/Yang Web: XML Syntax and RDF Semantics", WWW May 2002, Honolulu, Hawaii, USA. ACM Public.

Patel-Schneider P., Fensel D., 2002. "Layering the Semantic Web: Problems and Directions" - The Semantic Web-ISWC 2002: First International Semantic Web.

Rumbaugh J., Booch G., Jacobson I., 1999. "The Unified Modeling Language, Reference Manual", Addison-Wesley, New York (1999)

Sjachyn M., Beus-Dukic L., 2006. "Semantic component selection", 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems.