♦ Semantic Application Design

Philippe Larvet

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 postulate that this specification is made up of atomic requirements, logically linked together. On one hand, each requirement is expressed in natural language and this expression is seen as the semantic description of the requirement. On the other hand, the off-the-shelf components used to build the application can also be described through a semantic description, called a "semantic card." Within this context, we consider a component fulfills a requirement if the "semantic distance" between their two semantic descriptions is minimal. Consequently, building an application consists of building and optimizing the logical network of all the semantic optimal couples "requirement-component." The paper presents an automatic building and optimization 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. © 2008 Alcatel-Lucent

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 [1], 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." [18]

Many authors have described several methods to guide the building of component-based applications [2, 3, 10], but except within the field of semantic Web services composition (described in [14]), it seems that an automatic semantic-oriented process has not been considered as a serious approach to design. Traditional (i.e., nonsemantic) 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.

Moreover, it is important to notice that the research in semantic Web services [14, 16] does propose a semantic-oriented design approach, but this

Bell Labs Technical Journal 13(2), 75–92 (2008) © 2008 Alcatel-Lucent. Published by Wiley Periodicals, Inc. Published online in Wiley InterScience (www.interscience.wiley.com) • DOI: 10.1002/bltj.20304

Panel 1. Abbreviations, Acronyms, and Terms

2D—Two dimensional
3D—Three dimensional
comp—Component
FILO—First in, last out
GUI—Graphical user interface
HTML—Hypertext Markup Language
IMS—IP Multimedia Subsystem
IP—Internet Protocol
NL—Natural language
OWL-S—Ontology Web Language for Services
RDF—Resource Description Framework
RSS—Really Simple Syndication
semCard—Semantic card
semProx—Semantic proximity
semTag—Semantic tag
semVector—Semantic vector
synVector—Synonym vector
UML—Unified Modeling Language
URL—Uniform resource locator
VAT—Value added tax
WSDL—Web Service Description Language
XML—Extensible Markup Language

approach uses 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.

It is obvious that an application has to rely on a given specification. Here, we consider this specification exists under the form of natural language, informal text, which describes the functional and nonfunctional requirements that the application must address. 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, and
- The application design, i.e., the architecture of the solution, can be derived from the architecture of the problem, i.e., from the relationships between the requirements.
- Based on these three postulates, the objective of this paper is to propose a novel way to enhance the automatic production of a software application design.

The Proposed Process

We consider an application as a set of interrelated 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.

We consider that 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, graphical user interface (GUI) display mechanisms, text translation, Hypertext Markup Language (HTML) pages reading from uniform resource locators (URLs), and elementary functions for text processing. In addition to the concept of semantic component [8, 19, 24], we propose that each component was described through a "semantic card" which contains notably the "goal" of the component, expressed simply 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 that we explain in detail in the following, we propose to determine the "meaning" of the sentence representing the component's goal, i.e., its semantics, and to express it in terms of an appropriate computable data structure. Thus, the idea is to mark every "semantic atom" of functionality with its appropriate semantic data structure.

We also have a specification document containing the requirements that describe what the application will do, and what its functional and nonfunctional features are. The requirements are a set of sentences expressed in natural language. Each sentence has a meaning which can be discovered 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. This process is distinct from an ontology-based requirement analysis approach [8], where the authors propose a software requirements analysis method based on a domain ontology technique, in which a mapping can be established between a software requirements specification and several domain ontologies. Note that an ontology [21] is a formal description of the concepts manipulated in a given domain and of the relationships between these concepts. Here, no external ontology is used to help the requirements analysis, because the semantics are extracted from the text itself.

The set of sentences composing the requirements are logically linked to each other. Then, it is possible to determine a "requirement network" by scanning the links between the requirement atoms: this browsing will determine the structure of the "specification molecule"—the molecule that describes the problem.

Analyzing many specifications within the context of numerous industrial projects developed with an object-oriented approach [11] 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 a requirement are linked to each other when they both talk about a given datum, constraint, functionality, or feature of the targeted application. Then, the same link exists between the components implementing these requirements.

For example, imagine an application used to calculate the value added tax (VAT) of an invoice: the text of the specification contains two different paragraphs concerning the VAT computation. The first one explains the general method to compute the VAT. The second paragraph, perhaps located several pages later in the specification document, gives the different VAT rates according to the product categories. Obviously, these two pieces of requirement are linked together because they address the same data. And necessarily, the two design components implementing these requirements have to be linked together, because the computation of the VAT for a given product needs the general method to calculate a VAT amount.

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

Given these axioms, the proposed process consists of three steps:

1. Finding the components whose semantic distance is the shortest with the semantic atoms of requirements.
2. Organizing these components in order to constitute the "solution molecule," i.e., the initial architecture of the application. The initial design is made by replicating the problem molecule and using solution atoms instead of problem atoms, but these kinds of atoms are different and do not have exactly the same nature, so the initial component interaction model has to be optimized.
3. Optimizing the structure of the solution molecule in order to determine the best component interaction model.

This approach for composing software components is different from the usual processes described in literature, for instance, within the scope of semantic Web services automatic composition. McIlraith and Narayanan [14], for example, propose a composition and simulation solution using a Petri net formalism to create a composite Web service expressed in Ontology Web Language for Services (OWL-S) [23]. At the highest level, McIlraith and Narayanan start with a set of elementary Web services that have semantic descriptions, along with a specification of a desired goal to be satisfied by a composite Web service. The goal is expressed as a sentence written in the situation calculus language [17], i.e., a first-order logical language. They then describe how a composite Web service that satisfies the goal can be automatically constructed from the original set of given Web serices, if such a composite Web service exists. Petri net formalism is used to create the composite Web service, which is the final result specified using OWL-S. Within our approach, the initial component interaction model—that corresponds to the initial design of the future application—is not built from a generated structure that would come from the semantic descriptions of components; this initial model is built from the relationships between the application's requirements: an association between two requirements will

determine an association between the two components that cover these requirements. This point will be detailed later. **Figure 1** illustrates the principle of building a network that determines the "requirements molecule."

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 that a semCard has an Extensible Markup Language (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 whose name is mentioned in the semCard's header, and
3. A semantic tag (semTag) of the data, which represents a stereotype of a semantic data type and specifies the nature of the data; this semTag will be useful to determine and optimize the components' interactions and will be discussed in the second part of the paper.

The semantics of the operations' goals is defined with precise rules:

- 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
- The words composing the lists of concepts are defined in external dictionaries and belong to related domains that are referenced in the semCard.

These rules help to write operation goals that are terse and unambiguous.

Ontologies [21] could be used to summarize and formalize the definitions of concepts and domains, but this is not mandatory. Resource Description Framework (RDF) [15] or OWL [22] is 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 should also be suitable.

Panel 2 provides an example of the semCard for an RSS-feed-accessor component.

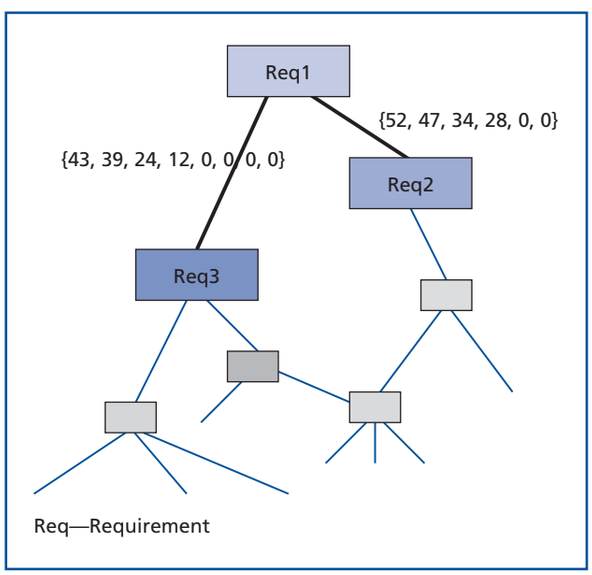

Figure 1.
Building the requirement network, i.e., the "problem molecule."

Determining the Meaning of Sentences

One of the key aspects of the process we propose here is the ability to compare the meaning of a requirement extracted from the specification document with a component's goal, which is part of the component's semCard. This is done to enable a mapping between components and the requirements they cover.

The key to this comparison is the ability to determine the "meaning" of a given text. We consider the meaning of a text 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 or concepts, and to be able to determine whether they are semantically close or not.

We now provide a brief overview of work related to semantic proximity (semProx) in natural language expressions. S. Khaitan and coauthors [9] provide a good summary of interesting work within this

Panel 2. 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>Deliver the titles of all the news of the RSS feed addressed by a given URL.</goal>
   <inputs>
    <input name="URL_RSS" concept="RSS#URL" semTag="URL" />
   </inputs>
   <output name="titles" concept="News#title" semTag="text" />
  </operation>
  <operation name="getDescriptionOfTitle">
   <goal>Deliver the description of the title of one news of the RSS feed addressed by a given
  URL.</goal>
   <inputs>
    <input name="URL_RSS" concept="RSS#URL" semTag="URL" />
    <input name="title"  concept="News#title" semTag="short_text" />
   </inputs>
   <output name="description_of_title" concept="News#description" semTag="text" />
  </operation>
 </component>
</semCard>
```

AQ 1

domain. Corley [4] presents a knowledge-based method for measuring the semantic similarity of texts; his method combines word-to-word similarity metrics, based on words' distances inside taxonomies, into a text-to-text metric. Guha and coauthors [7] present an application called Semantic Search, which is built on technologies including Web services and Semantic Web, which are creating a web of machine understandable data. They also provide an overview of an application framework upon which the Semantic Search is built. Mayfield and Finn [13] describe an approach to retrieval of documents containing both free text and semantically enriched markup. They present a prototype of a framework in which documents and queries can be marked up with statements in a specific semantic Web language. Guarino and coauthors [6] use linguistic ontology for content matching in information retrieval. Their approach applies only to the search field in a relevant class of information repositories, such as online yellow pages and product catalogs. Evans and Zhai [5] report on the application of a few noun-phrase analysis techniques to create indexing phrases for information retrieval. They describe a hybrid approach to the extraction of meaningful (continuous or discontinuous) subcompounds from complex noun phrases using both corpus statistics and linguistic heuristics.

The novelty of our approach is to propose a way to express the meaning of an elementary term in order to be able to process a comparison with another term. To do so, we build a "vector" with the synonyms of the term that can be found in a thesaurus. The concept of "vector" is not used here in the sense of a geometrical object, but as a close and limited set

of elements all having the same nature and all linked to the same equivalence relationship. We call it a "synonym vector" (synVector).

For example, the synVector of the term "battle" is:

battle = {fight, clash, combat, encounter, skirmish, scuffle, mêlée, conflict, confrontation, fracas, fray, action; struggle, crusade, war, campaign, drive, wrangle, engagement} (19)

Other examples are:

war = {conflict, combat, warfare, fighting, confrontation, hostilities, battle; campaign, struggle, crusade; competition, rivalry, feud} (13)

peace = {concord, peacetime, amity, harmony, armistice, reconciliation, ceasefire, accord, goodwill; agreement, pact, pacification, neutrality, negotiation} (14)

For synVectors, we define the following functions:

- synV(word) defines the synVector for the term "word,"
- card(V1) gives the cardinal of the vector V1, i.e., the number of synonyms inside V1,
- common(V1, V2) gives the synonyms that are common to V1 and V2, and
- avg(V1, V2) gives the average of the cardinals of V1 and V2.

For example, card(common(synV("battle"), synV ("war"))) 5 9. In other words, there are nine synonyms common to "battle" and "war."

We also define the concept of semantic proximity between two terms T1 and T2 by calculating a ratio taking into account the common synonyms within the two synVectors synV(T1) and synV(T2).

The semantic proximity is given by the formula:

semProx(T1, T2) = 100 * card(common(synV(T1), synV(T2))) / avg(synV(T1), synV(T2))

For example,

semProx("battle", "war") 5 100 * 9 / 0.5 * (19 + 13) 5 900 / 16 5 56.25.

In other words, in the union of the sets of synonyms for "battle" and "war," 56 percent of the elements are found to be duplicates.

The semProx expresses the proximity ratio between two terms. If semProx is greater than a given value A (for instance, 50) or close to 100, we consider the two terms to be semantically close. Inversely, if the semProx is less than a given value B (for instance, 10) or close to zero, the two terms are semantically distant. Values of levels A and B can be "tuned," according to the category of texts to be processed.

Of course, semProx("war", "peace") = 0 (the model seems to be consistent!)

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

- The sentence is analyzed and the pertinent words are extracted—nonpertinent words such as articles, prepositions, and conjunctions are ignored.
- For each pertinent word, a corresponding synVector is built.
- A global vector for the whole sentence that we call a "phraseVector" is built by assembling all the synVectors of the pertinent words contained in the sentence. This means a phraseVector is a vector of vectors, as shown in **Figure 2**.

For example, we will build a phraseVector for the following requirement, extracted from the specification of a call management system: "The caller

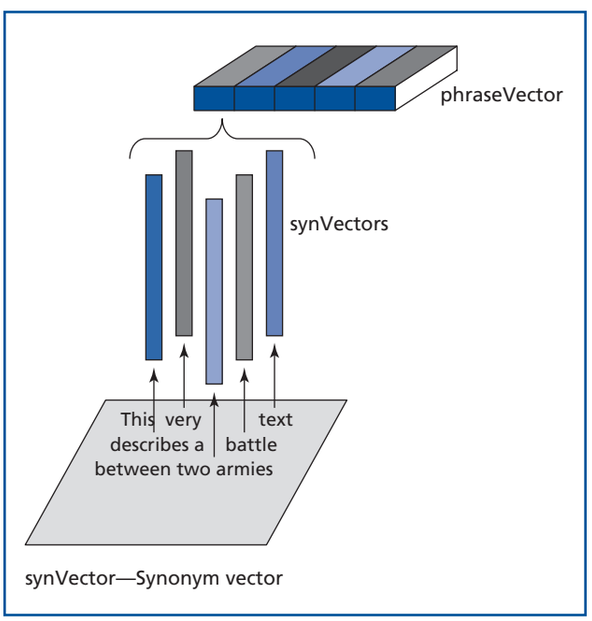

Figure 2.
Principle of building a phraseVector for a given sentence.

makes a call to a receiver by creating a message that contains the call subject, submitted to the receiver at the same time."

The pertinent terms are caller, call, make a call, receiver, message, subject, submit.

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

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

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

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

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

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

subject = {topic, theme, focus, subject matter, area under discussion, question, issue, matter, business, substance, text; field, study, discipline, area}(15)

submit = {offer, present, propose, suggest, tender}(5)

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

- The 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.
- The semantic proximity is calculated for each pair.
- The best values of the semProx among all comparisons are kept in an ordered external semantic vector (semVector), as a result of the comparison.
- The comparison of the semVectors for sentences S1 and S2, and S1 and S3, allows us to determine whether S1 is semantically closer to S2 or S3.

Figure 3 shows the principle of building a semVector from the synVectors of two sentences.

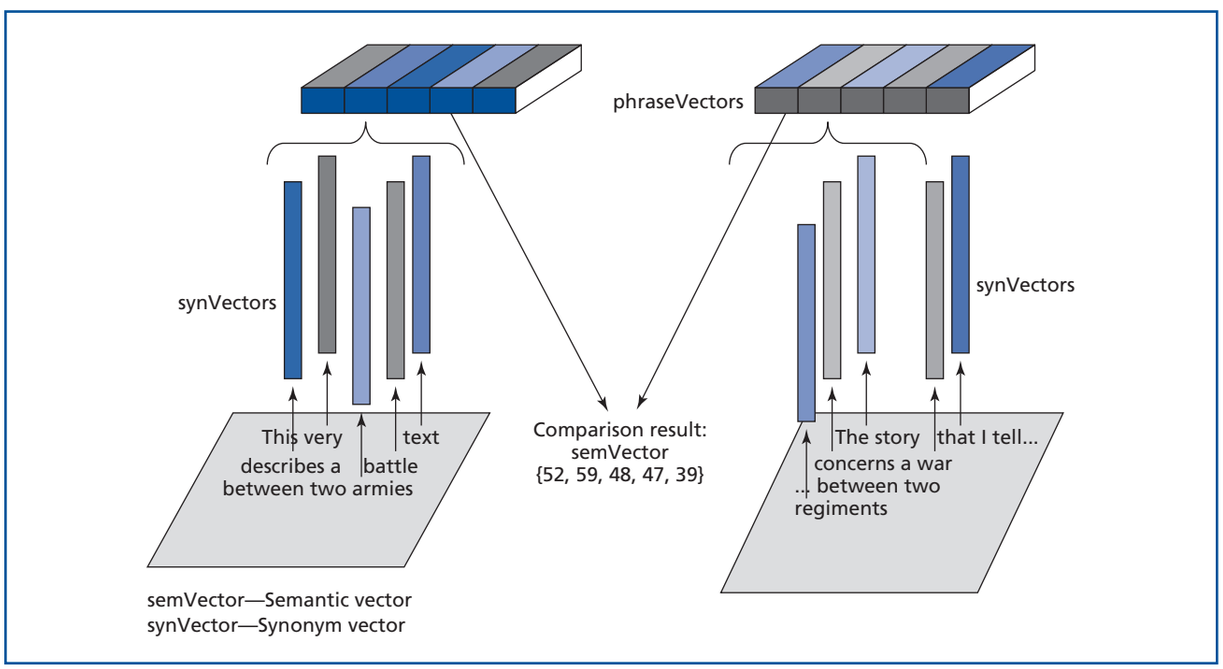

Figure 3.
Principle of building a semVector from the synVectors of two sentences.

The search for components that cover a given specification follows this approach:

- The phraseVectors of the sentences composing the specification are built and compared with the phraseVectors built from the goals of the components present in the component repository.
- The corresponding semVectors are built for every pair requirement-component.
- The best semVectors are kept and help to determine the components that are able to fulfill the requirements.

Table I shows an example of searching for correspondences between the requirement "The caller

Table I. SemVectors resulting from the comparison between the requirement sample and the components goals.

Component name	Component goal	semVector
AL Stock	Returns the value of Alcatel-Lucent's stock, in euros and dollars	{0, 0, 0, 0, 0, 0, 0}
AnDiscOrGeX	Analyzes, discovers, orchestrates, generates, and executes a composed service responding to a given user's request	{24, 15, 0, 0, 0, 0, 0, 0, 0}
Calculator	Returns the result of a two-terms operation among the four basic ones	{12, 0, 0, 0, 0, 0, 0}
Contact Info Searcher	Returns useful information concerning a person registered in a given X.500 directory	{21, 15, 0, 0, 0, 0, 0, 0}
DateTime	Returns the current date and time	{0, 0, 0, 0, 0, 0, 0}
Definition	Returns the English dictionary definition of a given word	{13, 0, 0, 0, 0, 0, 0}
FaxSender	Sends the given text of a fax to a given fax number	{35, 28, 0, 0, 0, 0, 0}
Language Finder	Determines the language in which a given text is written	{12, 0, 0, 0, 0, 0, 0}
MakeCall	Makes a call between two given phone numbers	{100, 100, 0, 0, 0, 0, 0}
Meteo	Returns weather information for a given city in France	{12, 0, 0, 0, 0, 0, 0}
Message Sender	Sends the text of a message to a given recipient	{100, 80, 80, 0, 0, 0, 0}
Phrase Vector Builder	Returns the phraseVector built from two different given sentences	{10, 0, 0, 0, 0, 0, 0}
RSS_Titles	Returns all the RSS titles for a given URL of an RSS feed	{0, 0, 0, 0, 0, 0, 0}
RSS_Description	Returns the description of a given RSS title for a given URL of an RSS feed	{0, 0, 0, 0, 0, 0, 0}
Semantic Discovery	Returns a list of the discovered services matching with a given list of concepts	{0, 0, 0, 0, 0, 0, 0}
Semantic Query Analyzer	Returns the list of the pertinent concepts extracted from a given phrase written in natural language	{10, 0, 0, 0, 0, 0, 0}
SMSSender	Sends a message as an SMS to a given mobile phone number	{100, 80, 0, 0, 0, 0, 0}
Synonyms	Returns the list of synonyms of a given word	{0, 0, 0, 0, 0, 0, 0}
Syn Vector Builder	Returns the synVector of a given sentence	{10, 0, 0, 0, 0, 0, 0}
Term Extractor	Returns the pertinent terms extracted from a given text	{12, 0, 0, 0, 0, 0, 0}
Translator	Returns the version of a given text translated into a given target language	{10, 0, 0, 0, 0, 0, 0}

RSS—Really Simple Syndication
semVector—Semantic vector
SMS—Short message service
synVector—Synonym vector
URL—Uniform resource locator

makes a call . . .” and a sample of a component set stored in a given component repository. The pertinent terms of the requirement are: `{caller, call, make call, receiver, message, subject, submit}`. The semVectors shown in the right column of Table I are the results of comparisons between the synVectors of these pertinent terms and those calculated from the component’s goal. A rapid view of these semVectors helps easily determine the components capable of fulfilling the requirement.

Determining the Problem Network

The requirement network summarizes and represents the links between the atoms of requirement that are the sentences composing the specification. **Figure 1** shows the principle of building the network that determines the specification molecule.

We use the semVector approach to reveal the links between the requirement atoms: the sentences of the specification are semantically compared two by two, by using the phraseVector plus semVector approach. Requirements are taken by pairs, phraseVectors are built, and semVectors are calculated. The result, for each requirement, is a set of vectors that represents the links, in terms of the 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. It means that each requirement has a limited number of other requirements that are semantically close. So, we consider that a requirement can be formally described, within the context of the whole specification, by a limited set of semVectors that represent the other requirements that are semantically closest.

The links can be represented in a two-dimensional (2D) or three-dimensional (3D) space, although they are really in an abstract multidimensional (multi-D) space. But the aim here is not to build a true network of the problem, which would be an objective difficult to reach, but only to model the problem, i.e., obtain a convenient representation on which we can think and communicate and that we can structurally compare to another. For example, **Figure 4** shows a model of a problem network as a convenient

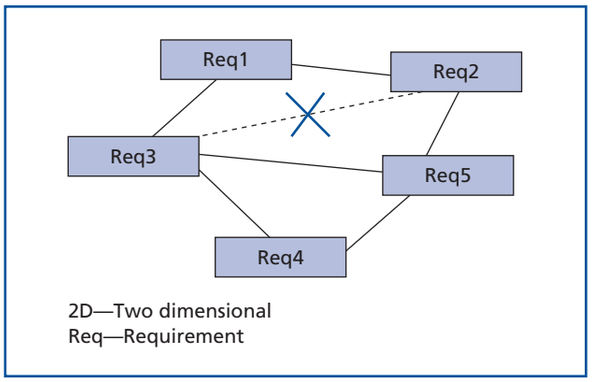

Figure 4.
Model of the problem network, showing a convenient 2D representation.

2D representation where links appear between some requirements. Only the best links are kept on this figure, i.e., the links whose semVector is larger.

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

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

only the link Req2–Req5 is 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. For example, we could consider that the link Req2–Req5 will be kept only if

```
diff(semVector(Req2, Req5), semVector(Req2, Req3)) > 10
```

In other cases, the limit for this diff() could be 5 or 15, depending on the problem category or the kind of requirements.

In reality, when building the architecture of an application, all the links whose `diff()` is greater than a minimal critical level could be kept in the problem model, and the optimization performed in the solution model, as discussed later.

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. The solution molecule has

the same spatial structure as the 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.

Having posed these assertions, 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 process:

1. Find the components that cover the requirements by using the semVector approach; this will build a list of components, not yet linked together. **Figure 5** illustrates this step.
2. 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, as illustrated in **Figure 6**.
3. Optimize this primary version in order to determine the best structure for the solution molecule. This will become the final architecture for the application.

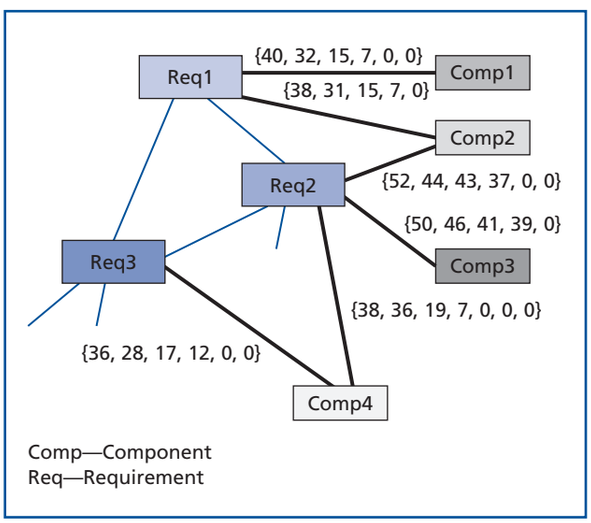

Figure 5.
Using the semVector approach to determine which components fulfill which requirements.

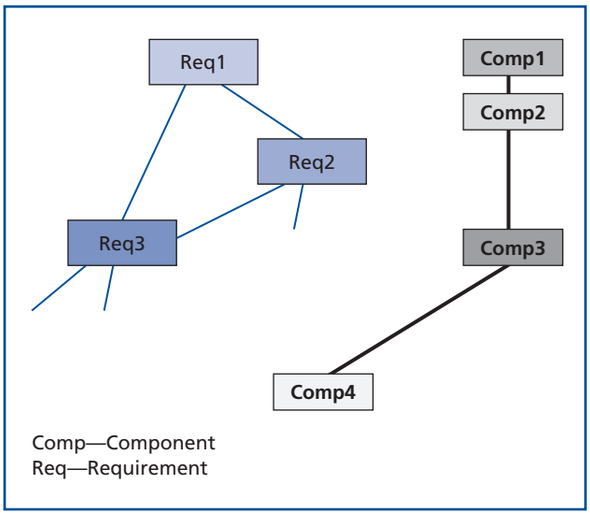

Figure 6.
Replicating the problem molecule structure inside the components list to get an initial version of the solution molecule.

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

Replicating the requirements links inside the component structure associates the components in the same way the 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 that the two corresponding components have an interaction together. 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 the two

components where the output of component 1 (Comp1) is an input for Comp2 or inversely.

Optimizing the Solution Network

The role of the optimization process is to determine the actual connections between the components in terms of actual data exchanges. To determine these connections automatically, we use the semantic tags added as semantic metadata to the inputs and output of the components' operations [12].

If these semantic annotations are suitably chosen and set, the components can be connected and their connectivity can be formally expressed. For example, if the output of Comp1.operation_A() semantically fits with the input of Comp2.operation_B(), then Comp1 can be connected to Comp2 through the link "output of A" to "input of B," and we are authorized to write something like:

```
out_A = Comp1.operation_A(A_parameters);
out_B = Comp2.operation_B(out_A);
```

or, more directly:

```
out_B = Comp2.operation_B(Comp1.oper-
ation_A(A_parameters));
```

This means the two connected data have the same semantic "dimension": i.e., they semantically fit each other—they are process-compatible; they share not only the same data type, but the same nature of data. This semantic data type is expressed by the semTag, similar to an Unified Modeling Language (UML) tagged value [18] and attached to the inputs and outputs within the semCards.

Here, a question can be asked: even if we can connect the output of Comp1 to the input of Comp2 because they semantically fit each other (for example, Comp1 produces a text and Comp2 consumes a text), how can we be sure that Comp2 is really waiting for the output of Comp1, instead of the output of Comp4, for instance, which is another component producing a text? In fact, we can be sure of the Comp1–Comp2 connectivity because the interactions are built by following the links that are present in the solution molecule as shown in Figure 6. Even if Comp4 produces a text, it is not directly linked to Comp2; consequently, there is no reason to try to combine their inputs to outputs.

SemTags ensure the consistency of component interfaces, and for this reason they are important elements for optimizing component interactions. For example, let us suppose `Comp1.operation_A()` provides a text, and `Comp2.operation_B()` is the operation `translate()` of a component Translator; it makes sense to translate a text; thus the output of `Comp1.operation_A()` has to fit with the input of `Translator.translate()`. But suppose `Comp1. operation_A()` provides the stock symbol for a given company. This symbol and the text taken as input by `Translator.translate()` can have the same data type (string). They are not semantically equivalent, because it does not make sense to try to translate a stock symbol. Therefore, the semantic information attached to these two data must be different, and consequently the two operations, and the two components, are not connectable.

Panel 3 provides an example of the semCard of the component translator, showing the useful semTags.

When components are Web services, for example, semCard descriptions can be generated from the Web Service Description Language (WSDL) [20]. But in order to set the semantic tags automatically, a specific semantic module can be used.

This module analyzes the names and types of the operation parameters, as described in WSDL, and searches for semantic correspondences in a specific ontology [12].

This ontology contains the links between the semantics of the current names and types of input and output data as they are usually used by programmers, and the corresponding semantic tags.

For example, data named "text" or "content" or "translated_page" or "description" with the type "string" will have the semantic tag "text" because the data has the "dimension" of a text. Data named "date" or "current_date," with a type "Date" or "String," will have the semantic tag "date."

This ontology can be expressed as a simple correspondence table, as shown in **Table II**.

Such an ontology is easy to build and to improve progressively by analyzing the contents of published components interfaces that show the practice of programmers and then by summarizing their good usages.

Panel 3. The SemCard of the Component Translator, Showing the Useful SemTags

```
<se  mCard>
 <URL>http://xxx.xx.xxx.x/components/Translation/Translator.asmx</URL>
 <component name="Translator">
  <domains>
   <domain name="Translation">
      <concepts list="translation, version, language, source language, target language, result" />
   </domain>
   <domain name="Text">
     <concepts list="text, chapter, paragraph, sentence, phrase, word, language" />
   </domain>
  </domains>
  <operation name="translate">
       <inputs>
       <input name="text_to_translate" concept="Text#Text" semtag="text" />
       <input name="source_language" concept="Translation#SourceLanguage"
       semtag="language" />
       <input name="target_language" concept="Translation#TargetLanguage" semtag="language" />
       </inputs>
       <output name="translated_text" concept=" Text#Text " semtag="text" />
       <goal>The goal of the operation is to provide a translated_text written in a given
       target_language as a result of the translation of a given text_to_translate written in a
       source_language>
       </goal>
  </operation>
 </component>
</semCard>
```

Table II. Ontology for automatically setting semantic tags within semantic cards.

Data name	Type	Semantic tag
text, content, page, description, ...	String	text
date, current_date, ...	String / Date	date
phone_number, mobile_phone, ...	String	telephone_ number
lang, language, dest_ lang, srce_ lang, ...	String	language
postal_code, zip_code, city_code, ...	String	zip_code
...		

Automating the Optimization of Solution Network

To describe the process that takes into account the semantic tags in order to build an automatic orchestration of components, consider the following example.

One of the requirements of a targeted application is "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 Really Simple Syndication (RSS)-accessor component and a translator component. The corresponding semCards for these components are shown in Panel 2 and Panel 3.

The RSS component is designed to gather information from RSS feeds accessible via the Internet, and its interface contains two operations: `getAllTitles()` obtains all the main titles of the feed for a given URL, and `getDescriptionOfTitle()` obtains 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 (out-

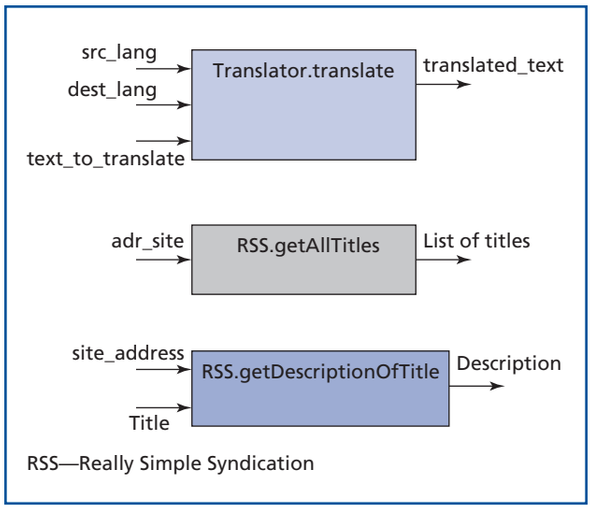

Figure 7.
How to automatically assemble three operations.

put) written in a destination language (input parameter).

Now, the problem is automatically and logically to assemble these two components, i.e., their three operations, as shown in **Figure 7**, in order to fulfill the original requirement: provide a translated version of a news feed.

First, consider semantic tags as inputs and outputs of operations, instead of data. Then, some possible connectivities appear as shown in **Figure 8**, but not precisely enough to make a fully consistent composition.

Second, review the main output of the targeted component assembly to determine which operations can provide its input, and to iterate the process for these operations—search for other operations that can provide their inputs. Then, go back progressively from the main output to the input data necessary to produce it, and, in the process, automatically assemble the different operations by linking their outputs and inputs.

At the same time, the links are stored in a first in, last out (FILO) stack under the form of pseudocode 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 desired: 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_translate, src_lang, dest_lang);
return translated_text;
```

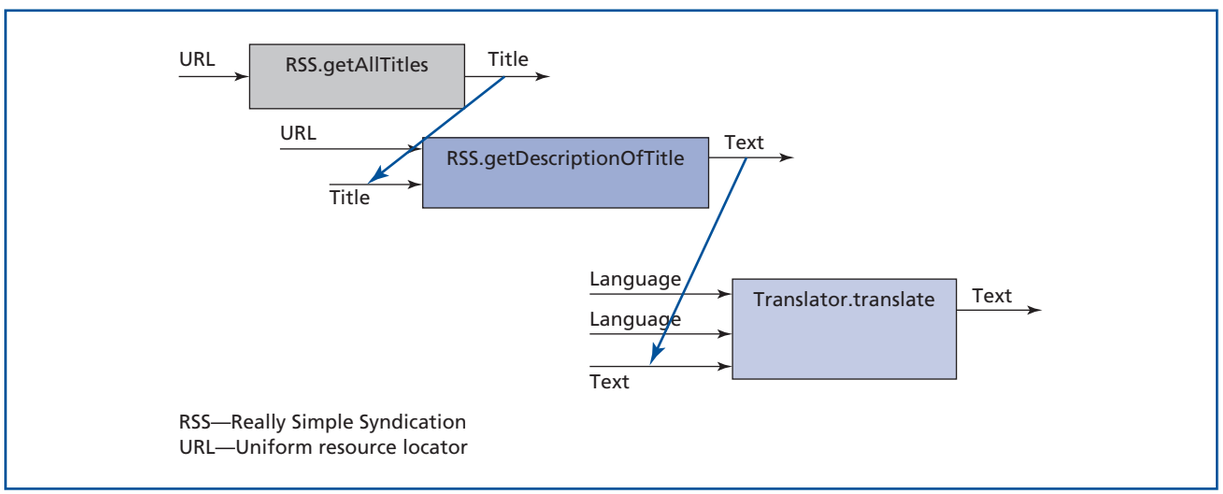

Figure 8.
Two possible connections—indicated by large arrows—appear by considering semantic tags instead of data names.

Refer again to the inputs of this operation, whose semantic tags are "language," "language" and "text." Data with a semantic tag "text" is provided by the operation RSS.`getDescriptionOfTitle()`.

This operation then can be connected to `Translator.translate()`. We can add the call to the operation `RSS.getDescriptionOfTitle()` in the stack, linking with `Translator.translate()` through the name of the exchanged parameter:

```
text_to_translate = RSS.getDescriptionOfTitle(site_address, title);
translated_text = Translator.translate(text_to_translate, src_lang, dest_lang);
return translated_text;
```

Now, refer to the inputs of `RSS.getDescriptionOfTitle()`, whose semantic tags are "URL" and "title." 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_address, title);
translated_text = Translator.translate(text_to_translate, src_lang, dest_lang);
return translated_text;
```

Having used and connected all the components allocated to the original requirement, the stack now contains the general texture of the component assembly, under the form of a nearly executable pseudocode. However, this pseudocode 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 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.
- 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 determine the source language of a given text automatically, or an operation `getSourceLanguage()` on the RSS feed component.

Panel 4 provides an example of a specific module, whose detailed description is outside the scope of this paper, which makes these refinements in order to complete the pseudocode.

This pseudocode 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

Panel 4. Refined Pseudo-Code

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

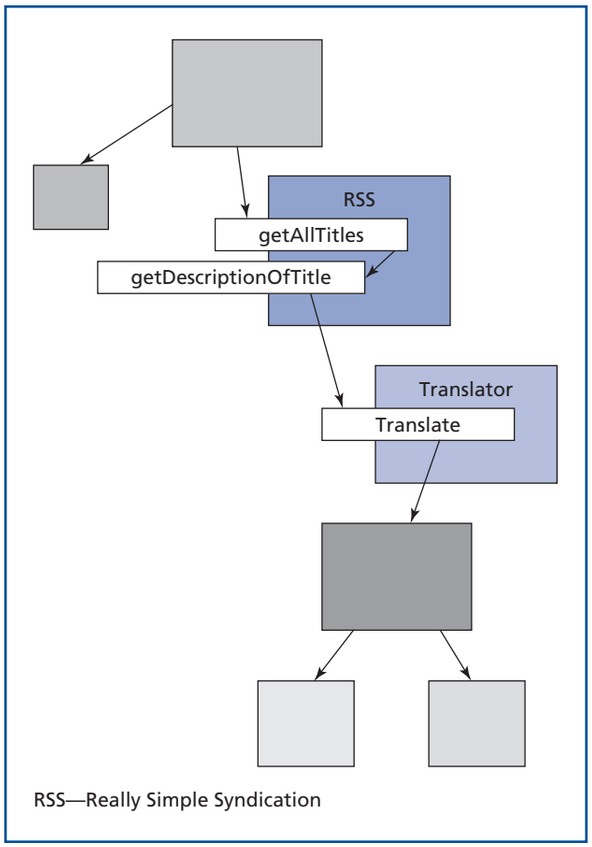

Figure 9.
Part of the final interaction diagram showing the assembly of the component translator and RSS.

process can be considered as a first draft of the design of the future application and is shown in **Figure 9**. Our intent is to deliver a draft with a quasi-executable pseudocode allowing validation tests of the architecture of the future application.

Conclusion

This paper has described an application of a natural language (NL) technology combined with a component-composition optimization process that enables automatic construction of software applications. We have presented an original but partly operational process to determine the meaning of NL text, and to use this meaning to find the right components to fulfill the original NL-expressed requirements of the application's specification. This process leads to an initial architectural structure of the targeted application, and we have presented a complementary process to optimize this structure to obtain an acceptable and testable draft of the application design.

The advantages of the proposed process are numerous: it is performed rapidly and fully automatically, it works directly from the original requirements of the application, and it delivers a quasi-executable pseudocode as a useful subproduct allowing validation testing of the architecture of the future application. Moreover, the traceability between the requirements and the architecture is certain, as another subproduct of the process.

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

Acknowledgments

The author would like to acknowledge the contributions of the following present members of the Villarceaux Research and Innovation IP Communication Applications project's IMS and Web Services (R&I-ICA-IWS) team: Fabien Balageas, Gérard Burnside, Olivier Le Berre, Philippe Jabaud, Patrick Fontaine for their useful comments, and Rick Hull, Ryan Skraba, anonymous reviewers, and Gail Burton-Dufay for their useful rereading and text improvements.

*Trademarks

Java is a trademark of Sun Microsystems Inc.

References

[1] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, Object-Oriented Analysis and Design with Applications, 3rd ed., Addison-Wesley, Upper Saddle River, NJ, 2007.

[2] F. Bordeleau and M. Hermeling, "Model-Driven Development for Component-Based Application Portability," COTS J., Aug. 2005, <http://www.cotsjournalonline.com/home/article.php?id=100379>.

[3] T. Chusho, H. Ishigure, N. Konda, and T. Iwata, "Component-Based Application Development on Architecture of a Model, UI and Components," Proc. 7th Asia-Pacific Software Engineering Conf. (APSEC '00) (Singapore, 2000), pp. 349–353.

[4] C. Corley and R. Mihalcea, "Measuring the Semantic Similarity of Texts," Proc. ACL Workshop on Empirical Modeling of Semantic Equivalence and Entailment (ACL '05) (Ann Arbor, MI, 2005), pp. 13–18.

[5] D. A. Evans and C. Zhai, "Noun-Phrase Analysis in Unrestricted Text for Information Retrieval," Proc. 34th Annual Meeting of the Assoc. for Computational Linguistics (ACL '96) (Santa Cruz, CA, 1996), pp. 17–24.
[6] N. Guarino, C. Masolo, and G. Vetere, "OntoSeek: Content-Based Access to the Web," IEEE Intelligent Syst., 14:3 (1999), 70–80.
[7] R. Guha, R. McCool, and E. Miller, "Semantic Search," Proc. 12th Internat. Conf. on World Wide Web (WWW '03) (Budapest, Hung., 2003), pp. 700–709.
[8] H. Kaiya and M. Saeki, "Ontology-Based Requirements Analysis: Lightweight Semantic Processing Approach," Proc. 5th Internat. Conf. on Quality Software (QSIC '05) (Melbourne, Austral., 2005), pp. 223–230.
[9] S. Khaitan, K. Verma, R. K. Mohanty, and P. Bhattacharyya, "Exploiting Semantic Proximity for Information Retrieval," Proc. Workshop on Cross Lingual Inform. Access, 20th Internat. Joint Conf. on Artificial Intelligence (IJCAI '07) (Hyderabad, India, 2007).
[10] M. Kirtland, Designing Component-Based Applications, Microsoft Press, Redmond, WA, 1998.
[11] P. Larvet, Analyse des Systémes: De l'Approche Fonctionnelle à l'Approche Objet, InterEditions, Paris, 1994.
[12] P. Larvet, "Composing Automatically Web Services Through Semantic Tags," Proc. 19th Internat. Conf. on Software and Syst. Engineering and Their Applications (ICSSEA '06) (Paris, Fr., 2006).
[13] J. Mayfield and T. Finin, "Information Retrieval on the Semantic Web: Integrating Inference and Retrieval," Proc. Semantic Web Workshop, 26th Internat. ACM SIGIR Conf. on Res. and Dev. in Inform. Retrieval (SIGIR '03) (Toronto, Can., 2003).
[14] S. Narayanan and S. A. McIlraith, "Simulation, Verification and Automated Composition of Web Services," Proc. 11th Internat. Conf. on World Wide Web (WWW '02) (Honolulu, HI, 2002), pp. 77–88.
[15] P. Patel-Schneider and J. Siméon, "The Yin/Yang Web: XML Syntax and RDF Semantics," Proc. 11th Internat. Conf. on World Wide Web (WWW '02) (Honolulu, HI, 2002), pp. 443–453.
[16] P. F. Patel-Schneider and D. Fensel, "Layering the Semantic Web: Problems and Directions," Proc. 1st Internat. Semantic Web Conf. (ISWC '02) (Sardinia, It., 2002), published in Lecture Notes in Comput. Sci. (LNCS 2342) (I. Horrocks and J. Hendler, eds.), Springer, Berlin, Heidelberg, New York, 2002, pp. 16–29.
[17] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, Cambridge, MA, 2001.
[18] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, MA, New York, 1999.
[19] M. Sjachyn and L. Beus-Dukic, "Semantic Component Selection—SemaCS," Proc. 5th Internat. Conf. on Commercial-Off-the-Shelf (COTS)-Based Software Syst. (ICCBSS '06) (Orlando, FL, 2006), pp. 83–89.
[20] World Wide Web Consortium, "Web Services Description Language (WSDL) 1.1," W3C Note, Mar. 15, 2001, <http://www.w3.org/TR/wsdl>.
[21] World Wide Web Consortium, "OWL Web Ontology Language: Overview," W3C Recommendation., Feb. 10, 2004, <http://www.w3.org/TR/owl-features/>.
[22] World Wide Web Consortium, "OWL Web Ontology Language: Semantics and Abstract Syntax," W3C Rec., Feb. 10, 2004, <http://www.w3.org/TR/2004/REC-owl-semantics-20040210/>.
[23] World Wide Web Consortium, "OWL-S: Semantic Markup for Web Services," W3C Member Submission, Nov. 22, 2004, <http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/>.
[24] H. Zhuge, "Semantic Component Networking: Toward the Synergy of Static Reuse and Dynamic Clustering of Resources in the Knowledge Grid," J. Syst. and Software, 79:10 (2006), 1469–482.

(Manuscript approved April 2008)

PHILIPPE LARVET is an engineer and technical

expert at the Alcatel-Lucent Bell Labs Research and Innovation Center in Villarceaux, France. His research topics include semantic definitions of Web services, model-driven application development, and natural language processing. Prior to his current assignment, he worked on introducing object technologies on E10 projects, and on knowledge management by developing tools and procedures to capture expertise and to enrich lexical dictionaries included in Alcatel's documentation system for

the Switching & Routing Division (SRD). Previously, he worked in several industrial companies as a software engineering expert in the scope of real-time and embedded systems, on numerous projects in object technologies at the level of specification and design. He is the author of several publications, among them two books, one regarding expert systems, and another covering systems analysis from the functional to the object-oriented approach.◆

Author Query

AQ1: Panels 2, 3, and 4: Please confirm that the indentations are as you want them to appear in the journal.