Searching and Matching Software Components with Multiple Interfaces

Luis Iribarne
Dpto. Lenguajes y Computación. Universidad de Almería
liribarn@ualm.es

Antonio Vallecillo
Dpto. Lenguajes y Ciencias de la Computación. Universidad de Málaga
av@lcc.uma.es

Abstract

Currently there is an increasing interest in the use of COTS components for building software applications. Component search and service matching are two of the issues involved in this process. Traditional proposals to deal with these problems are based on the simplistic assumption that components present only one interface with the services they offer, and that matching is done on a one-to-one basis. This work presents an extension of those approaches in which components offer several interfaces, and not only their *supported* services are contemplated, but also the external services they may require from other components to operate. In this context, some of the traditional component operators are extended – composition, substitutability, and equivalence– so they can deal with multiple interfaces. In addition, the problems arising in this new setting are discussed, and some of the possible solutions presented.

1 Introduction

In the last decade *Component-Based Software Engineering* is generating tremendous interest due to the development of plug-and-play reusable software, which has led to the concept of '*commercial off-the-shelf*' (COTS) components. Although currently more a goal to pursue a than a reality, this approach moves organizations from application *development* to application *assembly*. Constructing an application now involves the use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind. The ultimate goal, once again, is to be able to reduce developing times, costs, and efforts, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated.

Of course, this approach is challenging some of the current SE methods and tools. For instance, the traditional top-down development method based on successive refinements of the system requirements until a suitable concrete implementation of the final application's components is reached is no longer valid as such. In CBSE the system designer also has to have into account the specification of pre-developed COTS components that live in software repositories, that must be even considered when building the initial system's requirements in order to incorporate them into all phases of the development process [9, 11].

In this context, our particular long-term goal is to study the development of applications from COTS components, right from the specification of the application's software architecture. This specification describes the specification of *abstract* components, that may differ from the *concrete* specification of the COTS components residing in a given repository. In the simplest case each required service is separately specified, and each component implements just one

service. This simple case is the one considered by traditional approaches, and therefore the search and matching processes of components have been defined on a one-to-one basis [5, 15, 16]. However, this is not the common case in most real applications: in general, COTS components are coarse-grained components that integrate several services and offer several interfaces. Think for instance in an Internet navigator or a Word processor: apart from their core services they also offer many different ones, like web page composition, spell checking, etc.

This paper presents our ongoing work on these issues. Here we will study some of the problems that appear in this new setting, extending the traditional substitutability and equivalence operators to components that support multiple interfaces. In addition, we also take into account the services that components *require* from other components, not only the supported ones.

This paper is structured in 4 sections, the first one corresponds to this introduction. Section 2 presents our proposal, section 3 shows an example in order to illustrate it, and section 4 draws some conclusions and discusses future research lines.

2 Managing Components with Multiple Interfaces

In the first place, we need to define what we understand by a software component. Here we will adopt Clemens Szyperski's definition: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [12].

In addition, the "COTS" adjective will refer to a particular kind of component: a commercial entity (i.e. that can be sold or licensed) that allows for packaging, distribution, storage, retrieval and customization by users, which is usually coarse-grained, and lives in software repositories.

2.1 Interfaces

Components' capabilities and usages are specified by means of *interfaces*. An interface can be defined as "a service abstraction, that defines the operations that the service supports, independently from any particular implementation".

Interfaces can be described using many different notations, depending on the information that we want to include, and the level of detail of the specification. For instance, the interfaces of CORBA distributed objects are described using an Interface Description Language (IDL) which contains syntactic information about the objects' supported capabilities. Since CORBA follows an object-oriented model, CORBA interfaces consist of the object public attributes and methods. COM follows a similar approach, but components may have several interfaces, each one describing the signature of the supported operations. The new CORBA Component Model CCM [10] also contemplates that components may describe not only the services they support, but also the interfaces they require from others components during their execution.

Notice however that all this information remains at the *syntactic* level, that is, interfaces describe just the names of the methods, and the types of their arguments and return values. This information has proved to be insufficient for developing applications in open systems [14, 16]. In general, three main levels of component interoperability can be distinguished: the *signature* level, based just on the syntactic information of the components' supported and required services; the *protocol* level goes one step further, describing also the partial order in which components expect their methods to be called, and the order in which they invoke other components' methods; finally, the *semantic* level describe the "meaning" of operations and their expected behavior [13].

Of course, the notation used for describing components' interfaces will depend on the level we want to cover, and will also influence the sort of results that can be obtained when reasoning about the application's properties right from the specifications of the components' interfaces.

Current approaches at the signature level use IDLs for describing interfaces, which guarantee interoperability at this level among heterogeneous components, written in different languages, that use different object models, and that may be living in different machines, and using different execution environments. The IDLs defined by CORBA, COM and CCM are example of those.

At the protocol level most of the current approaches enrich the IDL description of the components interfaces with information about protocol interactions, using many different notations: finite-state machines [14], Petri nets [2], temporal logic [6, 7], or the π-calculus [3]. At this level the interfaces that a component offers or require from another are referred to as *roles*.

Finally, interfaces at the semantic level usually describe the operational semantics of components. Formal notations used range from Larch, with pre/post-conditions and invariants [4, 16], to algebraic equations [5], or the refinement calculus [8].

2.2 Interface Operators

Independently from the notation used, or the level of interoperability being dealt with, there are some commons operations that are of special interest when building applications from reusable components.

The first one is called *substitutability*, and refers to the ability of a component to replace another so that clients of the first one remain unaware of the change. This defines a partial order between components and is usually noted by "$\sqsubseteq$". With it, if we have a correct application, C is one of its constituent components, and $C \sqsubseteq D$, we could replace C with D in the application knowing that the application would continue working without problems.

At the signature level this operator just needs that supported methods of C are also supported by D. In case we also want to take into account the required operations, we would need to ask D not to use any external method not used by C. At the semantic level this operator is known as *behavioral subtyping* [1].

Based on operator "$\sqsubseteq$" (independently from the level it is defined) we can define an equivalence relation between interfaces, and say that two interfaces R_1 and R_2 are *equivalent* (noted by $R_1 \equiv R_2$) iff $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$.

A third operator defines when two components are *compatible* for interoperation (and is noted by $\rightleftarrows$). At the signature level this means that all exchanged messages are understood by each other, at the protocol level that their protocols match in each role they share [14, 3], and at the semantic level that invoked operations have the same meaning in the calling and in the called component.

Those operators are so important because they are the ones that service traders and brokers need to use in order to search for components in software repositories, from the abstract specifications of the components defined in the software architecture of an application. They are needed to guarantee that the composed application will respect the specification of the target application, and to prove the correctness of the final application.

So far those operators have been defined for components offering just one interface with their supported methods, and the search and matching processes of components have been based on them. Now we plan to extend them to components with multiple interfaces, both with the supported and required methods. This will allow us to define more realistic component search and matching processes. Therefore we define:

Definition 1 *A COTS component C will be determined by two sets of interfaces $C = \{\mathcal{R}, \overline{\mathcal{R}}\}$, the first one with the interfaces of the component's* supported *operations $\mathcal{R} = \{R_1, ..., R_n\}$, and the second one with the component's* required *operations $\overline{\mathcal{R}} = \{\overline{R}_1, ..., \overline{R}_m\}$.*

For simplicity we will write $C.\mathcal{R}$ and $C.\overline{\mathcal{R}}$ to refer to the two sets of interfaces of a given component C. On the other hand, components need to *compose* in order to build other components

and new applications:

Definition 2 (Component composition) *Given two components* $C_1 = \{\mathcal{R}_1, \overline{\mathcal{R}}_1\}$ *and* $C_2 = \{\mathcal{R}_2, \overline{\mathcal{R}}_2\}$*, its composition* $C_1 \mid C_2$ *is a new component* $C_3 = \{\mathcal{R}_3, \overline{\mathcal{R}}_3\}$*, such that*

$$\mathcal{R}_3 = \begin{cases} \mathcal{R}_1 \cup \mathcal{R}_2 & \textit{iff } \mathcal{R}_1 \cap \mathcal{R}_2 = \varnothing \\ \textit{undefined} & \textit{iff } \mathcal{R}_1 \cap \mathcal{R}_2 \neq \varnothing \end{cases} \qquad \overline{\mathcal{R}}_3 = \overline{\mathcal{R}}_1 \cup \overline{\mathcal{R}}_2 - \{\mathcal{R}_1 \cup \mathcal{R}_2\}$$

This operation is commutative and associative. We have defined it as a *partial* operation in order to avoid the conflicts that appear in an application in which two of its components offer the same service, i.e. *service overlaps*. This is for instance what happens in our PC when we install and activate two e-mail clients. In order to compose components in the presence of service overlaps we need a new operation for hiding services (eg. using *wrappers*):

Definition 3 (Hiding) *If* C_1 *is a component* $C_1 = \{\mathcal{R}_1, \overline{\mathcal{R}}_1\}$ *and* $\mathcal{R}$ *is a set of interfaces, we define a* hiding *operation "–" as:* $C_1 - \{\mathcal{R}\} = \{\mathcal{R}_1 - \mathcal{R}, \overline{\mathcal{R}}_1\}$

When composing components to build applications, we may also find that some of the services required by any of the components are missing in order to make the application work (i.e. *service gaps*). Hence we need to talk about "closures":

Definition 4 (Closure) *Let* C_1*,* C_2*, ...,* C_n *be a set of components, and* $A = C_1 \mid C_2 \mid ... \mid C_n$ *a new component obtained by composition. We shall say that* A *is* closed *iff* $\bigcup C_i.\overline{\mathcal{R}} \subset \bigcup C_i.\mathcal{R}$*.*

The following definitions will help us simplify the notation for relating interfaces, using traditional set notation:

Definition 5 (Interface Inclusion) *We shall say that* $\mathcal{R}_1 = \{R_1^1, ..., R_1^s\} \subseteq \mathcal{R}_2 = \{R_2^1, ..., R_2^t\}$ *if there exists a subset* $\mathcal{I} \subseteq \mathcal{R}_2$ *with* $\#(\mathcal{I}) = s$ *such that for all* $R \in \mathcal{R}_1$ *there exists one interface* $R' \in \mathcal{I}$ *with* $R \sqsubseteq R'$*.*

Please note that in this definition operator "$\sqsubseteq$" stands for the substitutability operator among simple interfaces, no matter the interoperability level it refers to (signatures, protocols or semantics). This operator overloading will be used throughout the paper, allowing the extension of the substitutability operator at all levels. The only difference will occur when considering its complexity: the operator for simple interfaces has a $O(1)$ complexity at the signature level, while it's exponential at the other two levels (cf. [3, 4]).

Definition 6 (Interface Intersection) *Interface intersection can be defined in the natural way, modulo interface equivalence:* $\mathcal{R}_1 = \{R_1^1, ..., R_1^s\} \cap \mathcal{R}_2 = \{R_2^1, ..., R_2^t\}$ *is the set of interfaces* $\mathcal{R} = \{R^1, \cdots, R^u\}$ *such that for all* $R^i \in \mathcal{R}$ *there exists one interface* $R_1^j \in \mathcal{R}_1$ *and another interface* $R_2^k \in \mathcal{R}_2$ *for which* $R^i \equiv R_1^j$ *and* $R^i \equiv R_2^k$*.*

Now we are in a position to extend the traditional substitutability operator to deal with components offering (and requiring) several interfaces:

Definition 7 (Component Substitutability) *Let* $C_1 = \{\mathcal{R}_1, \overline{\mathcal{R}}_1\}$ *and* $C_2 = \{\mathcal{R}_2, \overline{\mathcal{R}}_2\}$ *be two components. We shall say that* C_1 *can be* replaced *(or* substituted*) by* C_2*, and note by* $C_1 \sqsubseteq C_2$*, iff* $(C_1.\mathcal{R}_1 \subseteq C_2.\mathcal{R}_2) \wedge (C_1.\overline{\mathcal{R}}_1 \supseteq C_2.\overline{\mathcal{R}}_2)$

With this operator we can naturally extend the equivalence relation between components:

Definition 8 (Equivalence) *Two components* C_1 *and* C_2 *are said to be* equivalent $(C_1 \equiv C_2)$ *if they are mutually replaceable:* $C_1 \sqsubseteq C_2 \wedge C_2 \sqsubseteq C_1$*.*

2.3 Composing Components with Multiple Interfaces

Once all the concepts that define our working context have been defined, let us go back to the original problem: how to build applications from COTS components right from the specification of software architecture of the application.

In this setting, we need to confront the *abstract* specifications of the application's constituent components as described in the application's architecture $\mathcal{A}$, with the *concrete* specifications of the components that reside on a software repository $\mathcal{B}$. Our main goal here is the definition of a set of software components from the repository that offer the services defined by application $\mathcal{A}$. We shall call *configuration* to each one of those possible sets, imposing two conditions to any configuration S: (1) the services offered by the components of S must coincide with the services offered by application $\mathcal{A}$ (i.e. no service *gaps*), and (2) two components of the configuration will never provide a common service (i.e. no service *overlaps*).

In order to produce valid configurations for a given application $\mathcal{A}$, defined by a set of 'abstract' components $\{A_1, A_2, \cdots, A_n\}$, we have defined a three-step process: (1) *candidates* selection, (2) *configurations* generation, and (3) closure of configurations.

2.3.1 Selection of Candidate Components

The first step consists of selecting from the repository $\mathcal{B}$ those components $\{B_1, \cdots, B_m\}$ that may potentially participate in application $\mathcal{A}$ because they offer *at least one* of the services offered by the application. Therefore, we will concentrate just on the services supported by $\mathcal{A}$, without considering its required services. Thus, we can consider $\mathcal{A}$ as a single component $\mathcal{A} = A_1 \mid A_2 \mid \cdots \mid A_n$, and then we can talk about the set of interfaces $\mathcal{A}.\mathcal{R}$ and $\mathcal{A}.\overline{\mathcal{R}}$.

With this we can define the set of *candidate components* for application $\mathcal{A}$ with regard to repository $\mathcal{B}$ as: $C_{\mathcal{B}}(\mathcal{A}) = \{B \in \mathcal{B} \mid A.\mathcal{R} \cap B.\mathcal{R} \neq \varnothing\}$, i.e. those components in $\mathcal{B}$ that offer any service that $\mathcal{A}$ also offers.

In order to build this set we need to go through the repository only once, and decide for each of its elements whether it is a candidate or not. Hence, the complexity of this phase is $O(m)$, with $m = \#(\mathcal{B})$.

2.3.2 Generating Configurations

The second phase is to build a set $\mathcal{S}$ of all possible configurations with the candidate components. The basic idea is to build all combinations of candidate components (hiding the possible service overlaps), and select those combinations $Sol = \{S_1, \cdots, S_l\}$ such that $A.\mathcal{R} = Sol.\mathcal{R}$ (hence guaranteing no service gaps).

A *backtracking* algorithm to do that is shown in figure 1. It produces, from the candidates set $C_{\mathcal{B}}(A) = \{C_1, \cdots, C_k\}$, and from the application $\mathcal{A}$ considered as one component, a set $\mathcal{S}$ of valid configurations. For this algorithm, the initial invocation is $\mathcal{S} = \varnothing$; $Sol = \varnothing$; `configs`$(1, Sol, \mathcal{S})$.

As we can see, the algorithm explores all possibilities, building a final set with all valid configurations (line 8). Each individual configuration is generated by trying all candidates, incorporating those that offer services in $\mathcal{A}$ not yet considered (line 6). For the way the algorithm works, no service gaps or overlaps may happen, hence producing only valid configurations. The order of complexity of this algorithm is $O(2^k)$, with k the number of candidate components, i.e. $k = \#(C_{\mathcal{B}}(A))$.

2.3.3 Closing configurations

Once all configurations have been generated, we need to *close* them in order to get complete applications. The process of closing a given configuration can be carried out by applying any of

```
1   function configs(i, Sol, S)
2     /* 1 ≤ i ≤ k is the level, Sol the configuration being built */
3     if i ≤ k then
4       // case 1: try to include C_i or part of it in Sol
5       if C_i.R ∩ Sol.R ≠ ∅ then // C_i or part of it can be included
6         Sol := Sol ∪ {C_i − {Sol.R ∩ C_i.R}};
7         if A.R ⊆ Sol.R then // if Sol is a valid configuration...
8           S := S ∪ {Sol} // ...we include it in S
9         else // but if still we have service gaps...
10          configs(i + 1, Sol) // ...keep on searching
11        endif;
12        Sol := Sol − {C_i − {Sol.R ∩ C_i.R}}
13      endif;
14      // case 2: try a configuration without C_i
15      configs(i + 1, Sol)
16    endif
17  endfunction
```

Figure 1: Algorithm for generating valid configurations.

the existing algorithms for calculating the transitive closure of a set (i.e. a configuration) with regard to another bigger set (in this case the repository $\mathcal{B}$), and goes beyond the scope of this paper.

2.4 Further considerations about the configurations

Once we count with a process that produces a set of valid *configurations* that can implement the services described in an application's software architecture, there are some issues that may need to be stressed.

2.4.1 Metrics for configurations

The process shown here builds a set of valid configurations so that the system designer can choose the one that fits better his/her requirements. A good idea could be assigning (somehow) *weights* to configurations that help the user to finally select one. Those weights could include many different factors, from commercial issues (component prices or availability), to complexity (number of supported and required interfaces), etc.

Defining weights would also allow an additional benefit: we could change the algorithm into a 'branch and bound' one and use upper bounds to prune many of the options in the exploration tree, hence notably improving the execution time of the algorithm.

2.4.2 Matching the application's internal structure

Configurations have been built from the existing candidate components in the repository, but without having into account the application's internal structure, as defined by its software architecture. By structure we mean the divisions of the applications services in abstract components: $A = \{A_1, \cdots, A_n\}$. But this issue can also be contemplated in our proposal. It is simply the case of discarding those configurations with components that crosscut the boundaries established by the architecture. This can be defined as follows:

Definition 9 *Let* $A = \{A_1, \cdots, A_n\}$ *be an application, and* $S = \{S_1, \cdots, S_m\}$ *a valid configuration for* A*, i.e. a set of components from the repository that offer the same services as* A *offers, and with no service overlaps. We shall say that* S respects A*'s internal structure iff* $\forall\, i \in \{1..m\}, \forall\, j \in \{1..n\} \bullet S_i.\mathcal{R} \cap A_j.\mathcal{R} \neq \varnothing \Rightarrow (S_i \sqsubseteq A_j) \vee (A_j \sqsubseteq S_i)$

This definition forces components of S either to 'contain' or to 'be contained' in the components of A, but respecting their 'boundaries'.

3 An Example Application

In this section we will present an example to illustrate the concepts introduced in this paper. It consists of a simple desktop application E, with some basic components:

$$E = \{\textit{Calculator (CAL), Calendar (CIO), Agenda (AG), Meeting Scheduler (MS)}\}$$

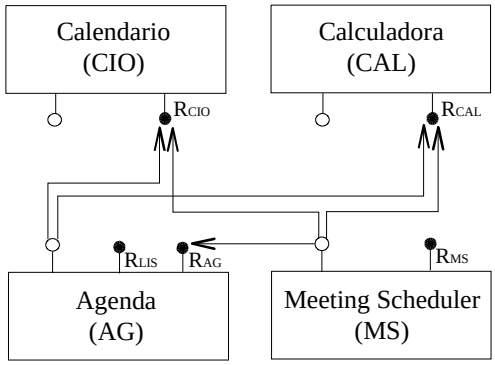

Figure 2: The architecture of example application E.

Figure 2 shows the application's architecture, by means of component decomposition and the interconnections among those components. Black circles represent supported interfaces, while white circles represent required ones. Arrows represent invocations. In addition, users may interact with each of those components through their supported interfaces too. The definition of those components using our notation is described in table 1, left column.

Let's try to use our process now, starting from the selection of the candidate components from a given software repository. The first thing to do is to consider the application E as a single component, obtaining $E.\mathcal{R} = \{R_{CIO}, R_{CAL}, R_{AG}, R_{LIS}, R_{MS}\}$, $E.\overline{\mathcal{R}} = \{\}$.

With those services we can go through the components in the repository, selecting those that offer at least one of the services that E supports. A possible result of this match is shown in the right column of table 1. In this example, six components have been found as candidates. Please notice that candidate component C_6 requires an external service defined by interface R_P. In case this component is included in a configuration, we would need to close it first in order to produce a working application. The third step of our process would take care of this, closing the configurations with regard to repository $\mathcal{B}$.

Table 2 shows the behavior of algorithm `configs()` for the example. In this case the algorithm generates $2^6 = 64$ combinations, since the number of candidates was 6. In the table, the first column is the sequential number of the combination, columns 2-7 show the service provided by each component, and column 8 contains the configuration produced (if any).

Please note how the algorithm tries to include the services not previously included, until all the services in the application are covered, or the candidate list is empty. If a combination does not cover all the application's services, it is discarded (eg. combinations 3, 4, 63 and 64).

E Architecture (Abstract specifications)	$C_{\mathcal{B}}(E)$: Set of candidate components (Concrete specifications)
$CIO = \{R_{CIO}\}$ $CAL = \{R_{CAL}\}$ $AG = \{R_{AG}, R_{LIS}, \overline{R}_{CAL}, \overline{R}_{CIO}\}$ $MS = \{R_{MS}, \overline{R}_{AG}, \overline{R}_{CAL}, \overline{R}_{CIO}\}$	$C_1 = \{R_{CIO}\}$ $C_2 = \{R_{CAL}\}$ $C_3 = \{R_{AG}, R_{CIO}, \overline{R}_{CAL}\}$ $C_4 = \{R_{LIS}\}$ $C_5 = \{R_{MS}, R_{AG}, \overline{R}_{CIO}\}$ $C_6 = \{R_{CAL}, R_{LIS}, \overline{R}_P\}$

Table 1: Abstract component specifications, vs. concrete candidate component specifications.

	C_1	C_2	C_3	C_4	C_5	C_6	Configuración
1	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	-	C_1, C_2, C_3–$\{R_{CIO}\}$, C_4, C_5–$\{R_{AG}\}$
2	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	-	Same as above
3	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	-	-	$\varnothing$ – R_{MS} missing (gap)
4	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	-	-	$\varnothing$ – R_{MS} missing (gap)
5	R_{CIO}	R_{CAL}	R_{AG}	-	R_{MS}	R_{CAL}	C_1, C_2, C_3–$\{R_{CIO}\}$, C_5–$\{R_{AG}\}$, C_6–$\{R_{CAL}\}$
6	R_{CIO}	R_{CAL}	R_{AG}	-	R_{MS}	-	$\varnothing$ – R_{LIS} missing (gap)
...							...
9	R_{CIO}	R_{CAL}	-	R_{LIS}	R_{MS}, R_{AG}	-	C_1, C_2, C_4, C_5
13	R_{CIO}	R_{CAL}	-	-	R_{MS}, R_{AG}	R_{LIS}	C_1, C_2, C_5, $C_6 - \{R_{CAL}\}$
17	R_{CIO}	-	R_{AG}	R_{LIS}	R_{MS}	R_{CAL}	C_1, C_3–$\{R_{CIO}\}$, C_5–$\{R_{AG}\}$, C_6
21	R_{CIO}	-	R_{AG}	-	R_{MS}	R_{CAL}, R_{LIS}	C_1, C_3–$\{R_{CIO}\}$, C_4, C_5–$\{R_{AG}\}$, C_6–$\{R_{LIS}\}$
25	R_{CIO}	-	-	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	C_1, C_4, C_5, C_6–$\{R_{LIS}\}$
29	R_{CIO}	-	-	-	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	C_1, C_5, C_6
33	-	R_{CAL}	R_{CIO}, R_{AG}	R_{LIS}	R_{MS}	-	C_2, C_3, C_4, C_5–$\{R_{AG}\}$
37	-	R_{CAL}	R_{CIO}, R_{AG}	-	R_{MS}	R_{LIS}	C_2, C_3, C_5–$\{R_{AG}\}$, C_6–$\{R_{CAL}\}$
49	-	-	R_{CIO}, R_{AG}	R_{LIS}	R_{MS}	R_{CAL}	C_3, C_4, C_5–$\{R_{AG}\}$, C_6–$\{R_{LIS}\}$
53	-	-	R_{CIO}, R_{AG}	-	R_{MS}	R_{CAL}, R_{LIS}	C_3, C_5–$\{R_{AG}\}$, C_6
...							...
63	-	-	-	-	-	R_{CAL}, R_{LIS}	$\varnothing$ – R_{CIO}, R_{AG}, R_{MS} missing (gap)
64	-	-	-	-	-	-	$\varnothing$ – All services missing (gap)

Table 2: Result of the `configs()` algorithm for the example application.

For this example 12 configurations are found, three of them closed (1, 9 and 33), and 4 of them that respect the internal architecture of the application (1, 5, 13 and 21). From here, it is a decision of the system's designer which configuration to use.

It is important to observe that the process described here has been defined for complete applications. However, it could also be used for some parts of an application, too. In this way we could allow the designer to decide which parts of the whole application he wants to implement with COTS components from the repository, and which parts not, applying the process only to the selected parts.

4 Conclusions y future work

This is part of an ongoing work that tries to build applications using COTS components, from the specification of the application's software architecture. This paper presents two main contributions. First, it extends traditional interface operators to the case in which components support more that one interface, and that also specify the interfaces required from other components to work. Those operators are the ones that service traders can use for searching and matching components in software repositories. In addition, we have been very careful with those extensions so they can extend the traditional interoperability operators no matter the level they

refer to (signatures, protocols, or semantics). Our second contribution is an algorithm for producing *configurations* based on the previous operators, ie. collection of COTS components from a repository that fulfill the application's requirements, and that do not have either service gaps or service overlaps.

There is still a lot of work ahead. First, we want to use formal notations for describing the applications' software architecture, using ADLs like Darwin, Rapide, or LEDA. We also want to enrich current IDLs to cope with protocols or semantic information compatible with the ADLs used. And once we have compatible notations for describing the *abstract* and the *concrete* specification of components, we plan to extend current repositories and service traders so they can make effective use of all this information. Finally, we also plan to define some metrics and heuristics for configurations, which can help systems designers in their decision processes when building applications from existing COTS components.

References

[1] P. America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages*, pages 60–90. LNCS 489, Springer Verlag, 1991.

[2] R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 474–494. Springer-Verlag, 1999.

[3] C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending CORBA interfaces with π-calculus for protocol compatibility. In *Proc. of TOOLS'33*, France, June 2000. IEEE Computer Society Press.

[4] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pages 258–267, Berlin, Germany, 1996. IEEE Press.

[5] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, and V. Berzins. Software component search. *Journal of Systems integration*, 6:93–134, September 1996.

[6] J. Han. Semantic and usage packaging for software components. In A. Vallecillo, J. Hernández, and J. M. Troya, editors, *Proc. of the ECOOP'99 Workshop on Object Interoperability (WOI'99)*, pages 25–34, June 1999.

[7] D. Lea. Interface-based protocol specification of open systems using PSL. In *Proc. of ECOOP'95*, number 1241 in LNCS. Springer-Verlag, 1995.

[8] A. Mikhajlova. *Ensuring Correctness of Object and Component Systems*. PhD thesis, Åbo Akademi University, October 1999.

[9] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Trans. on Software engineering*, 21(6):528–562, June 1995.

[10] Object Management Group. *The CORBA Component Model*, June 1999. http://www.omg.org.

[11] S. Robertson and J. Robertson. *Mastering the Requirement Process*. Addison-Wesley, 1999.

[12] C. Szyperski. *Component Software*. Addison-Wesley, 1998.

[13] A. Vallecillo, J. Hernández, and J. M. Troya, editors. *Proc. of the ECOOP'99 Workshop on Object Interoperability*, June 1999.

[14] D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.

[15] A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170, April 1995.

[16] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, October 1997.