Selecting Software Components with Multiple Interfaces

Luis Iribarne
University of Almería
Escuela Politécnica. 04120 Almería, Spain
liribarne@ual.es

José M. Troya and Antonio Vallecillo
University of Málaga
ETSI Informática. 29071 Málaga, Spain
{troya,av}@lcc.uma.es

Abstract

Component-based software development is gaining recognition as the key technology for the construction of high-quality, evolvable, large software systems in timely and affordable manners. Component search and service matching have become two of the key issues involved in this process. However, current proposals addressing these issues are based on the simplistic assumptions that components present only one interface with the services they offer. This work presents an extension of those approaches in which components may offer and require several interfaces, extending the traditional component "substitutability" operator. In addition, an algorithm for selecting COTS components with multiple interfaces from a repository in order to implement a given software architecture is presented.

1. Introduction

Component-Based Software Development (CBSD) is generating tremendous interest due to the need of plug-and-play reusable software for developing applications, 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*. In CBSD, constructing an application involves the use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind. The ultimate goal is to be able to reduce development 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.

In CBSD, the system designer has to take into account the specification of pre-developed COTS components available in software repositories, that must be even considered when building the initial requirements of the system, incorporating them into all phases of the development process [17, 21].

In this approach, an abstract software architecture of the system is defined first, that describes the specification of *abstract* components and their relationships. These abstract components are then matched against the list of *concrete* COTS components available in a repository. This process produces a list of the candidate components from the repository that could form part of the application: both because they provide some of the services required by the application, and because they fulfil some of the user's (non-functional) requirements such as price, security, etc. With this list, the system architecture is re-examined in order to accommodate as much candidates from the list a possible.

Additionally, *wrappers* may be used for adapting the COTS components selected (for hiding extra services non required, or for adapting their interfaces for compatibility or interoperability), and some *glue* language can be also used for composing and coordinating component interactions.

Traditionally, the search and matching processes of components have been defined on a one-to-one basis [9, 29, 30]. However, this is not the common case in most real applications: COTS components are coarse-grained components that integrate several services and offer many interfaces. For instance, an Internet navigator or a Word processor, apart from their core services, they also offer others, such as web page composition, spell checking, and so on.

This paper presents a proposal to deal with these issues. In particular, we study some of the problems that appear in this new setting—such as service *gaps* and *overlaps*—extending the traditional compatibility and substitutability operators to deal with components that support multiple interfaces. In addition, we also take into account the services that components *require* from other components, not only the services they support, according to the currently widely-accepted component-oriented programming paradigm [24].

This paper is structured into five sections. After this introduction, Section 2 presents an overview of the COTS based software applications. Next, Section 3 presents our proposal and an example that illustrates it. After that, Section 4 discusses some related work, and finally Section 5 describes some conclusions, and future research activities.

2. Software components and interfaces

In the first place, let us 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" [24]. In addition, the "COTS" qualifier refers to a particular kind of component: a commercial entity that allows for packaging, distribution, storage, retrieval and customization by users, which is usually coarse-grained, and lives in software repositories [4, 18, 22, 26].

2.1. Component Interfaces

Component capabilities and usages are specified by *interfaces*. For our proposal, an interface is "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. The interfaces describe just the names of the methods, the types of their arguments, and the return values—i.e., the signature. However, this information has proved to be insufficient for developing applications in open systems [28].

On top of signatures, the *semantic level* deals with the "meaning" of operations—i.e, the behaviour [6, 13]—and the *protocol* level deals just with the components' service access protocols—i.e., the partial order in which components expect their methods to be called, and the order in which they invoke other methods [25].

These three interoperability levels are the ones considered in this paper for selecting and matching components. Of course, the notation used for describing component 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.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 [28], Petri nets [3], temporal logic [10], or the π-calculus [5]. At this level the offered/required interfaces are referred to as *roles*. Finally, interfaces at the semantic level usually describe the operational semantics of components. Formal notations used range from the Larch family of languages, that use pre/post-conditions and invariants [8], to algebraic equations [9], or the refinement calculus [16].

2.2. Interface operators

Independently from the notation or the level of interoperability used, there are some common 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 operator defines a partial order between components and is usually noted by "$\leqslant$". With it, given an application and two components C and D, $D \leqslant C$ means that we can replace C with D in the application with the warranty that the application will continue working without problems.

At the signature level, substitutability $D \leqslant C$ roughly a matter of checking that all services offered by C are also offered by D. At the protocol level, we need to check two main issues: (a) that all messages accepted by C are also accepted by D, and that outgoing messages of D are a subset of the outgoing messages of C; and (b), we need to test that the relative order among the incoming and outgoing messages of both components are consistent [5, 28]. Finally, semantic substitutability is known in this context as "*behavioural subtyping*". Here, the behaviour refers to the specification on how the object methods manipulate the object attributes, and behavioural types are defined as an extension of object signature types that associate behaviour to signatures and to identify subtypes that conform to their supertypes not only syntactically, but also *semantically*.

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

A third operator defines when two components are *compatible* for interoperation (and is noted by "$\bowtie$"). At the signature level this means that all exchanged messages are understood by each other, and at the protocol level that their protocols match in each role they share [28, 5]. At the semantic level, it implies that the behaviour provided by a component should be accordant to the behaviour expected from its client component, as has been the basis for the "design by contract" development discipline [14].

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, looking for those components that can substitute the specification of *abstract* required components defined in the software architecture of an application.

Throughout this paper, operator "$\leqslant$" will stand for the substitutability operator at any level. However, special care should be taken when considering the implementation and complexity of operator "$\leqslant$" at the different levels: it has a $O(1)$ complexity at the signature level, while it is exponential at the other two levels (cf. [5, 8]).

2.3. Extending the “$\leqslant$” operator

Traditionally, the substitutability operator has been defined for components offering just one interface with their supported methods, and the search and matching processes of components have been based on it. Now we plan to extend this operator for components with multiple interfaces, including both supported and required. This will allow us to define component search and matching processes.

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

For simplicity we will write $C.\mathcal{R}$ and $C.\overline{\mathcal{R}}$ to refer to the two sets of interfaces. For interoperability at the signature level, R_i's and $\overline{R}_j$'s represent standard interfaces (e.g., CORBA or COM interfaces) composed just of a set of public attributes and methods. At the protocol level, each R_i or $\overline{R}_j$ describes a ‘role’, and at the semantic level, each of them corresponds to a description of an interface decorated with semantic information (e.g., with pre/post conditions).

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

Definition 2 (Interface inclusion) *Let $\mathcal{A} = \{A_1, ..., A_s\}$ and $\mathcal{B} = \{B_1, ..., B_t\}$ be two sets of interfaces. We shall say that $\mathcal{A} \subseteq \mathcal{B}$ if for all $A_i \in \mathcal{A}$ there exists one interface $B_j \in \mathcal{B}$ with $B_j \leqslant A_i$.*

Please note that in this definition, operator “$\leqslant$” stands for the substitutability operator among simple interfaces, no matter the interoperability level it refers to (signatures, protocols or semantics). Likewise, interface intersection can be defined in the natural way:

Definition 3 (Interface intersection) *Let $\mathcal{A}$ and $\mathcal{B}$ be two sets of interfaces. We define $\mathcal{R} = \mathcal{A} \cap \mathcal{B}$ as the set of interfaces $\mathcal{R} = \{R_1, \cdots, R_u\}$ such that for all $R_k \in \mathcal{R}$ there exists one interface $A_i \in \mathcal{A}$ and another interface $B_j \in \mathcal{B}$ for which $R_k \leqslant A_i$ and $R_k \leqslant B_j$ ($A_i \equiv B_j$).*

On the other hand, components need to be *composed* in order to build other components and applications:

Definition 4 (Component composition) *Let $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ and $C_2 = (\mathcal{R}_2, \overline{\mathcal{R}}_2)$ be two components. We define $C_1 \mid C_2$ as a component $C_3 = (\mathcal{R}_3, \overline{\mathcal{R}}_3)$, such that*

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

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

Please note that the composition 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*. In order to compose components we need a new operation for hiding services:

Definition 5 (Hiding) *Let C_1 represent a component $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ and let $\mathcal{R}$ be a set of interfaces. The* hiding *operator “$-$” is defined by $C_1 - \{\mathcal{R}\} = (\mathcal{R}_1 - \mathcal{R}, \overline{\mathcal{R}}_1)$, allowing us to hide from C_1 all interfaces in $\mathcal{R}$.*

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 “closed” applications, with no service gaps:

Definition 6 (Closed application) *Let C_1, C_2, ..., C_n be 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 $\bigcup_{i \in \{1..n\}} C_i.\overline{\mathcal{R}} \subseteq \bigcup_{i \in \{1..n\}} C_i.\mathcal{R}$.*

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

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

3. Building software applications from COTS

Once all the concepts that constitute 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 the application architecture.

Definition 8 (Application Architecture) *An application architecture $\mathcal{A}$ will be determined as a set of component-abstract specifications $\mathcal{A} = \{A_1, \ldots, A_n\}$, such that A_i can be described as a COTS component C in Definition 1.*

3.1. An example application

In this section we will present an example to illustrate the following concepts. The example consists of a simple desktop application E, with some basic components: a Calculator, a Calendar, an Agenda, and a Meeting Scheduler.

Figure 1 shows the application architecture, expressed in terms of its components and their interconnections. Black circles represent supported interfaces, while white circles represent required ones. Arrows represent invocations. The definition of those components using our notation is described in Table 1—left column—which shows their dependencies, too.

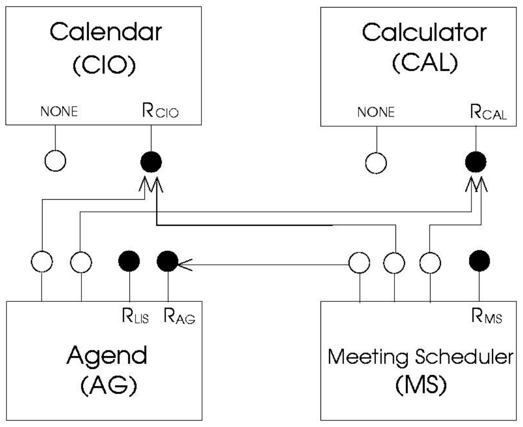

Figure 1. Architecture of example application.

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

Table 1. Abstract vs. concrete specifications

3.2. The building process

In this setting, we need to confront the *abstract* specifications of the application components—as described by the architecture $\mathcal{A}$—with the *concrete* specifications of the components in a software repository $\mathcal{B}$. For this we define:

Definition 9 (Configuration) *Given an application architecture $\mathcal{A}$, and a repository of components $\mathcal{B}$, we define a* configuration *S as a set of software components from $\mathcal{B}$, fulfilling the following two conditions: (a) the set of services offered by the components of S must coincide with the set of services offered by $\mathcal{A}$ (i.e., no service* gaps*), and (b) no two components from S provide the same common service (i.e., no service* overlaps*).*

Let us see in more detail how configurations can be built. The following three-step process has been defined in order to produce the set of all valid configurations for a given application $\mathcal{A}$: (a) Selection of components from the repository that match any of the services defined in the architecture; (b) generation of configurations with no service gaps or overlaps; and (c), closure of the configurations obtained. Let us describe all these steps in more detail.

3.2.1 Component Selection.

The first step consists of selecting from the repository $\mathcal{B}$ the set of components $B = \{B_1, \ldots, B_m\}$ that may potentially participate in application $\mathcal{A}$ because they offer *at least one* of the services offered by the application.

Definition 10 (Candidates) *Let $\mathcal{A} = \{A_1, \cdots, A_n\}$ be an application architecture and let $\mathcal{B} = \{B_1, \ldots, B_m\}$ be the repository. We define the set of candidate components as $C_{\mathcal{B}}(\mathcal{A}) = \{B \subset \mathcal{B} \mid \mathcal{A}.\mathcal{R} \cap B.\mathcal{R} \neq \varnothing\}$.*

Therefore, we will concentrate just on the services supported by $\mathcal{A}$, by considering $\mathcal{A} = A_1 \mid A_2 \mid \ldots \mid A_n$ as a new component with $\mathcal{A}.\mathcal{R}$ and $\mathcal{A}.\overline{\mathcal{R}}$ to refer the set of provided and required interfaces, respectively.

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. If $m = card(\mathcal{B})$ is the number of components in the repository, and L the complexity of the substitutability operator (constant for signatures, exponential for protocol and semantic checks), then the complexity of the selection process is $O(mL)$.

Let us try to use the example application E, 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 last step of our process takes care of this, closing the configurations with regard to repository $\mathcal{B}$.

3.2.2 Generating the configurations.

The second phase tries 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 in order to compose them without problems), and select those combinations $Sol = \{S_1, \cdots, S_l\}$ such that $\mathcal{A}.\mathcal{R} \subseteq Sol.\mathcal{R}$ (hence guaranteeing no service gaps).

Table 2 shows a backtracking algorithm that implements this process. It produces, from the set of candidates $C_{\mathcal{B}}(\mathcal{A}) = \{C_1, \cdots, C_k\}$, and from the application $\mathcal{A}$ considered as one component, a set $\mathcal{S}$ of valid configurations.

The initial invocation of the algorithm is $\mathcal{S} = \varnothing$; $Sol = \varnothing$; `configs`$(1, Sol, \mathcal{S})$. An implementation of this algorithm in C++ is available at `http://www.cotstrader.com/selectingCOTS/home.html`.

As we can see, the algorithm explores all possibilities, building a final set with all valid configurations step by step (line 11). Each individual configuration (line 9) is generated by trying all candidates, incorporating those services $\mathcal{C}_i.R_j$ not yet included in $\mathcal{A}$, and discarding those already considered (lines 8 and 10). When the algorithm finishes, variable $\mathcal{S}$ contains all configurations. Due to the way the algorithm works, no service gaps or overlaps may happen, hence producing only valid configurations.

```
 1  function configs($i, Sol, \mathcal{S}$)
 2    // $1 \leq i \leq size(C_{\mathcal{B}}(A))$ traverse the repository
 3    // $Sol$ is the configuration being built
 4    // $\mathcal{S}$ contains the set of all valid configurations $\mathcal{A}$
 5    if $i \leq size(C_{\mathcal{B}}(A))$ then
 6      for $j := 1$ to $size(C_i.\mathcal{R})$ do // all service in $C_i$
 7        // we try to include $C_i.R_j$ service in $Sol$
 8        if $\{C_i.R_j\} \cap Sol.\mathcal{R} = \varnothing$ then // $C_i.R_j \notin Sol.\mathcal{R}$?
 9          $Sol := Sol \cup \{C_i.R_j\}$;
10          if $\mathcal{A}.\mathcal{R} \subseteq Sol.\mathcal{R}$ then // Is $Sol$ a configuration?
11            $\mathcal{S} := \mathcal{S} \cup \{Sol\}$; // if so, it is included in $\mathcal{S}$
12          else // but if there are still service gaps ...
13            configs($i, Sol, \mathcal{S}$); // search in $C_i$ ...
14          endif
15          $Sol := Sol - \{C_i.R_j\}$;
16        endif
17      endfor
18      configs($i+1, Sol, \mathcal{S}$); // Next in $C_{\mathcal{B}}(A)$.
19    endif
20  endfunction
```

Table 2. Obtaining all valid configurations.

For instance, Table 3 shows some results produced by algorithm `configs()` for the desktop E. In this case the algorithm generates $2^9 = 512$ combinations, since the total of services provided by all candidate components was 9. From those 512 possible combinations, only 16 configurations are valid, which are shown in the table. In addition, some other discarded configurations are shown for completeness.

3.2.3 Closing configurations.

Once all configurations have been generated, we need to *close* them in order to get a "complete" application. The process of closing a given configuration can be carried out by applying any of 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 $\mathcal{B}$). This process is well-known and established, and is beyond the scope of this paper.

Following with our desktop example, in Table 3 columns 2-7 show the services provided by each component in each configuration, column 8 describes the configuration, in terms of its constituent components (hidding also the appropriate services), and column 9 shows an "R" if the configuration respects the application structure, and a "C" if it is a closed configuration.

For instance, configuration number 1 contains all candidate components but C_6, and each component provides just one service. This configuration is closed and respects the application's structure. From the 16 configurations found, four are closed, and four respect the structure. Now it is a decision of the system designer to select the configuration that better suits his requirements from this list of valid configurations, or to revisit the original architecture.

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.

3.3. Further considerations

Once we count with a process that produces a set of valid *configurations* that can implement the services described in an application's architecture, let us discuss some issues which are worth pointing out.

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 *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), or other non-functional requirements [1, 7].

Defining weights could also bring additional benefits. First, we could order the configurations obtained using their weights, ranking them according to the user's given criteria. And second, we could change the algorithm into a 'branch and bound' one that use upper bounds to prune many of the options in the exploration tree.

On the other hand, configurations have been built from the existing candidate components in the repository, but without taking into account the application's internal structure—as defined by its application architecture.

By structure we mean the divisions of the applications services in abstract components $\mathcal{A} = \{A_1, \cdots, A_n\}$. But this issue can also be contemplated in our proposal. It is simply the case of discarding those configurations with components that cross the boundaries established by the architecture, i.e., those that do not *respect* the architecture $\mathcal{A}$.

	C_1	C_2	C_3	C_4	C_5	C_6	Configurations	
–	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	-	-	NONE: R_{MS} missing (gap)	
1	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	-	$C_1, C_2, C_3\text{–}\{R_{CIO}\}, C_4, C_5\text{–}\{R_{AG}\}$	RC
2	R_{CIO}	R_{CAL}	R_{AG}	-	R_{MS}	R_{LIS}	$C_1, C_2, C_3\text{–}\{R_{CIO}\}, C_5\text{–}\{R_{AG}\}, C_6\text{–}\{R_{CAL}\}$	R
3	R_{CIO}	R_{CAL}	-	R_{LIS}	R_{MS}, R_{AG}	-	C_1, C_2, C_4, C_5	C
4	R_{CIO}	R_{CAL}	-	-	R_{MS}, R_{AG}	R_{LIS}	$C_1, C_2, C_5, C_6\text{–}\{R_{CAL}\}$	R
5	R_{CIO}	-	R_{AG}	R_{LIS}	R_{MS}	R_{CAL}	$C_1, C_3\text{–}\{R_{CIO}\}, C_4, C_5\text{–}\{R_{AG}\}, C_6\text{–}\{R_{LIS}\}$	R
6	R_{CIO}	-	R_{AG}	-	R_{MS}	R_{CAL}, R_{LIS}	$C_1, C_3\text{–}\{R_{CIO}\}, C_5\text{–}\{R_{AG}\}, C_6$	
–	R_{CIO}	-	R_{AG}	-	-	R_{LIS}	NONE: R_{CAL}, R_{MS} missing (gaps)	
7	R_{CIO}	-	-	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	$C_1, C_4, C_5, C_6\text{–}\{R_{LIS}\}$	
8	R_{CIO}	-	-	-	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	C_1, C_5, C_6	
–	R_{CIO}	R_{CAL}	-	-	-	-	NONE: R_{AG}, R_{LIS}, R_{MS} missing (gaps)	
9	-	R_{CAL}	R_{AG}, R_{CIO}	R_{LIS}	R_{MS}	-	$C_2, C_3, C_4, C_5\text{–}\{R_{AG}\}$	C
10	-	R_{CAL}	R_{AG}, R_{CIO}	-	R_{MS}	R_{LIS}	$C_2, C_3, C_5\text{–}\{R_{AG}\}, C_6\text{–}\{R_{CAL}\}$	
–	-	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	-	NONE: R_{CIO} missing (gap)	
11	-	R_{CAL}	R_{CIO}	R_{LIS}	R_{MS}, R_{AG}	-	$C_2, C_3\text{–}\{R_{AG}\}, C_4, C_5$	C
12	-	R_{CAL}	R_{CIO}	-	R_{MS}, R_{AG}	R_{LIS}	$C_2, C_3\text{–}\{R_{AG}\}, C_5, C_6\text{–}\{R_{CAL}\}$	
–	-	R_{CAL}	-	-	R_{MS}	-	NONE: R_{CIO}, R_{AG}, R_{LIS} missing (gaps)	
13	-	-	R_{AG}, R_{CIO}	R_{LIS}	R_{MS}	R_{CAL}	$C_3, C_4, C_5\text{–}\{R_{AG}\}, C_6\text{–}\{R_{LIS}\}$	
14	-	-	R_{AG}, R_{CIO}	-	R_{MS}	R_{CAL}, R_{LIS}	$C_3, C_5\text{–}\{R_{AG}\}, C_6$	
15	-	-	R_{CIO}	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	$C_3\text{–}\{R_{AG}\}, C_4, C_5, C_6\text{–}\{R_{LIS}\}$	
16	-	-	R_{CIO}	-	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	$C_3\text{–}\{R_{AG}\}, C_5, C_6$	
–	-	-	-	-	-	R_{CAL}, R_{LIS}	NONE: R_{CIO}, R_{AG}, R_{MS} missing (gaps)	

Table 3. Some results of the `configs()` algorithm for the example.

4. Related Work

The contributions presented in this paper can be related to three main research lines, which work on component interoperability, component acquisition, and about building systems from commercial components.

In the first place we find those works that address component interoperability, at any of the three levels: signature level [20, 29], protocol level [5, 28], and semantic level [1, 2, 8, 10]. These works have been already discussed in Section 2, and our approach tries to extend them to deal with components offering (and requiring) multiple interfaces.

The second group of work is related to the search and selection processes of components from software repositories, also known as components acquisition. These works take into account the architecture requirements—denoted in some cases as *applications engineering*—and the component specifications available in well-known software repositories—denoted in some cases as *domain engineering*. In that sense, there are a lot of works on component acquisition [18, 19, 15], here we underline three of them.

For instance, Rolland [22] proposes a technique by which requirements are captured by transition diagrams called maps, based in four basic models: the As-Is model, the To-Be model, the COTS model, and the integrated match model. However, this approach does not propose any particular way for specifying COTS components, or gives any indication on how to carry out the syntactic and semantic matchmaking process between components. Another proposal is due to Goguen et al. [9], where a set of criteria for searching and selecting components from a repository is presented and discussed. However, this proposal deals only with components offering simple interfaces, and therefore the problems of service gaps and overlaps do not appear. Furthermore, we highlight an important work: Seacord et al. [23] propose a process for identifying component ensembles that satisfy a system requirements specification based on a knowledge base of system integration rules.

Finally, in the third place we underline those works on building applications from COTS components. In this case, we highlight two interesting work areas. First one is the COTS-Based Systems (CBS) Initiative[1] at Software Engineering Institute (SEI), where we find the work of Kurt Wallnau, Scott Hissam and Robert Seacord on building systems from commercial components [27]. In second place, we find another very interesting proposals about CAFE and ESAPS projects[2], both from European Software Institute (ESI). One of their works is due to Cherki et al. [6], where they are describing a platform called Thales for building systems based on COTS parts.

[1]Available at `http://www.sei.cmu.edu/cbs` web site.

[2]These projects are available at `http://www.esi.es/Cafe` and `http://www.esi.es/esaps` web sites.

5. Conclusions and future work

This paper presents two main contributions. First, it extends traditional interface operators to the case in which components support more than one interface and also specify the interfaces required from other components to work. We have been very careful when defining those extensions so they can extend the traditional interoperability operators no matter the level they refer to (signature, protocols, or semantics). And second, we have shown an algorithm for producing *configurations* based on the previous operators.

There are several possible extensions of our work, such as defining some metrics and heuristics for configurations, which can help systems designers in their decision processes when building applications from existing COTS components. On the other hand, our work is a first step towards a more ambitious goal of defining a methodology for the development of applications by assembling COTS components living in software repositories (or in the Web), whose composition implements a target software architecture. In this sense, we want to use formal notations for describing the application software architecture, e.g., by using an Architectural Description languages (ADL) such as Darwin, LEDA, ACME, or UML-RT. Then, we also need to enrich current IDLs to cope with protocols or semantic information compatible with the ADLs used, beyond the mere signature information they contain now. And once we have compatible notations for describing the *abstract* and the *concrete* specification of components, we plan to work on extension of current repositories and service traders so they can make effective use of all this information. Some preliminary results have already been obtained, and can be found in [12].

References

[1] C. F. Alves et al. Using non-functional requirements to select components: A formal approach. In *(IDEAS'01)*, 2001.

[2] P. America. Designing an object-oriented programming language with behavioral subtyping. Number 489 in LNCS, pages 60–90. Springer-Verlag, 1991.

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

[4] A. W. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, Sep-Oct 1999.

[5] C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending CORBA interfaces with π-calculus for protocol compatibility. In *TOOLS'00*, pages 208–225. IEEE Press, 2000.

[6] S. Cherki et al. Development Support prototype for system families based on COTS. Technical report, ESAPS Project, 2001. http://www.esi.es/esaps.

[7] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.

[8] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE-18*, pages 258–267. IEEE Press, 1996.

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

[10] J. Han. Semantic and usage packaging for software components. In *WOI-ECOOP'99*, pages 25–34, 1999.

[11] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering. Putting the Pieces Together*. Addison-Wesley, May 2001. ISBN 0-201-70485-4.

[12] L. Iribarne, J. M. Troya, and A. Vallecillo. Trading for COTS components in open environments. In *27th Euromicro*, pages 30–37. IEEE CS Press, 2001.

[13] G. T. Leavens and M. Sitaraman. *Foundations of Component-Based Systems*. Cambridge University, 2000.

[14] B. Meyer. *Object-Oriented Software Construction. 2nd Ed.* Series on Computer Science. Prentice Hall, 1997.

[15] B. C. Meyers and P. Oberndorf. *Managing Software Acquisition*. SEI Series in SE. Addison-Wesley, 2001.

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

[17] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Trans. SE*, 21(6):528–562, 1995.

[18] C. Ncube and N. Maiden. COTS software selection. In *Continuing Collaborations COTS Development*, 2000.

[19] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, pages 115–117, March 2001.

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

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

[22] C. Rolland. Requirements engineering for COTS based systems. *IS Technology*, 41:985–990, 1999.

[23] R. C. Seacord, D. Mundie, and S. Boonsiri. K-BACEE: Knowledge-Based Automated Component Ensemble Evaluation. In *27th Euromicro*. IEEE CS Press, 2001.

[24] C. Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[25] A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP'2000 Workshop Reader*, number 1964 in LNCS. Springer-Verlag, 2000.

[26] K. C. Wallnau, D. Carney, and B. Pollack. How COTS software affects the design of COTS-intensive systems. SEI Interactive, 1998.

[27] K. C. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Addison-Wesley, 2002.

[28] D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. Prog. Lang. Syst.*, 19(2):292–333, Mar. 1997.

[29] 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, Apr. 1995.

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