View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional de la Universidad de Almería (Spain)

Semantic Matching of Components at run-time in Distributed Environments

Javier Criado, Luis Iribarne, Nicolás Padilla, Rosa Ayala

Applied Computing Group, University of Almería, Spain
{javi.criado,luis.iribarne,npadilla,rmayala}@ual.es

Abstract. Software factories are a key element in Component-Based Software Engineering due to the common space provided for software reuse through repositories of components. These repositories can be developed by third parties in order to be inspected and used by different organizations, and they can also be distributed in different locations. Therefore, there is a need for a trading service that manages all available components. In this paper, we describe a matching process based on syntactic and semantic information of software components. This matching operation is part of a trading service which is in charge of generating configurations of components from architectural definitions. With this aim, the proposed matching allows us to evaluate and score the possible configurations, thus guiding a search process to build the architectural solution which best fulfills an input definition.

Keywords: components, matching, reuse, trading, heuristics, run-time

1 Introduction

Software reuse is a topic of ongoing interest in the construction of applications, especially in the component-based development. In this sense, *Component-Based Software Engineering* (CBSE) provides mechanisms for building applications from the union of pieces [5]. Certain types of component-based software systems have the need of performing a dynamic management of the elements which can be part of the applications [6]. In such cases, components are used for building or adapting software applications at run-time. In this sense, when a new architectural solution is needed, the most appropriate elements are selected from a set of available components. The selection of components involved in this process requires the existence of accessible repositories which can be inspected and queried in order to calculate the best possible configuration.

Repositories can be stored locally or can be intended for public use and shared by different organizations in a distributed environment [15] [16]. This scenario is usual in systems that build their applications using components developed by third parties, for example, based on COTS (*Commercial Off-The-Shelf*) [1]. Thus, these repositories constitute the existing market of components from which the software is built. These repositories of components can be managed similarly to a service directory, which are accessed by certain entities for offering services,

and by other entities for making use of the available services. With this aim, *trading* techniques are useful to facilitate the execution of export and import operations of services [11]. Furthermore, trading mechanisms can be used to solve component-based architectures from an architectural definition [10].

In this paper, we present a semantic matching mechanism applied in a search algorithm for construing architectures of software components at run-time. This operation is used as part of a trading service which manages repositories of components developed by third-parties. Specifically, the managed elements are coarse-grained COTS components, which can be specified by the DSL (Domain-Specific Language) shown in Figure 1. This language distinguishes between two levels of representation: *abstract* and *concrete*. Abstract components are used for describing architectural definitions (*i.e.*, the set of features that an architecture must include) whereas concrete components are utilized for defining architectural solutions (*i.e.*, the characteristics of an architecture consisting on real software components). Each specification is divided into four parts, with the aim of describing *functional*, *extra-functional*, *packaging* and *marketing* information.

The object implementing this trading service, named as *Semantic Trader*, is in charge of building architectural solutions from the input information contained in architectural definitions. With this regard, matching operations between abstract and concrete components are performed at run-time for scoring and evaluating the different configurations of components that are taken into account as possible solutions. Both, the matching operations and the trading service are part of a methodology for adapting architectures at run-time [3, 4].

The semantic matching of the proposed trading service is based on the following assumption: the possible types that can be used for the description of the inputs and outputs of the interfaces' operations are restricted. Therefore, we propose to create a *namespace* that groups all possible types, which are iden-

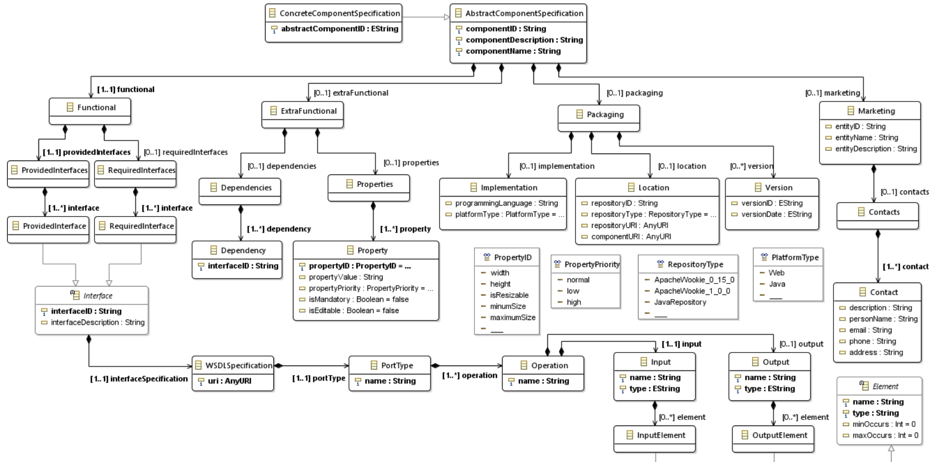

Fig. 1. Specifications of components

tified as `trader:typeName`. These types are described using an XML schema syntax, and are referenced from the definition of interfaces, in the corresponding WSDL [8] fragment of the model that contains the specification of the component (see Figure 1). These types are equivalent to complex data types that provide: (a) information about the name, the type and the cardinality of the elements composing the complex data type, and (b) information about the operations using this data type and if it is used as input or output. In addition to the description of interfaces, semantic information is present in the evaluation of components and architectures used in the heuristics of an A* search algorithm used for generating the architectural solutions.

The remainder of this paper is organized as follows. Section 2 describes the semantic matching approach used for building the architectural solutions at run-time. Section 3 discusses some related work. Finally, Section 4 presents the conclusions and the future work.

2 Semantic Matching of the Trading Service

The final goal of the proposed trading service is to build architectural solutions at run-time. This action is based on a semantic matching between the components of an input architectural definition and the different configurations which are evaluated in a search process. This search is performed using an A* algorithm. In this type of algorithms, a graph represents the search space and its nodes identify the states to advance in the search. The goal is to find the least-cost path to the target node from a starting node. Cost calculation is made using an evaluation function $f(x) = g(x) + h'(x)$. Function $g(x)$ represents a known distance (pre-calculated) between the initial node and the current node. Furthermore, $h'(x)$ identifies the estimated value of an admissible heuristic ($h(x)$) concerning the distance from the current node to the target node. In order to be admissible, the heuristic should not overestimate the real value of the calculated distance.

This type of algorithm always find a solution if one exists. In addition, the search process should not necessarily explore all nodes of the graph to find this solution. The explored search space and, consequently, the complexity of the algorithm depends on the quality of the heuristics. In the worst case, the order is exponential, whereas the order of the best case (where the estimated heuristic is close to the optimal case) is linear. Another reason for choosing this type of algorithm is the run-time nature of the calculation of configurations. The exploration path always moves towards a solution whose distance from the target node is lower than the previous state. Therefore, we can keep a reference to the last 'best solution' and make use of it if the trading service is forced to finish the search (for example, due to time constraints or other restrictions).

In our proposal, each graph node represents a configuration of concrete components, so that, a node is adjacent to another if its configuration differs in one component. Thus, each iteration of the A* search algorithm is executed until a configuration that meets the architectural definition is found. The *Semantic Trader* is in charge of executing this algorithm, evaluating each configuration

and building the architectural solution. With this aim, the proposed mechanism for generating the best concrete architecture is based on the following operations: (a) select the candidates, (b) calculate the configurations, (c) close the configurations, (d) calculate the configuration which are compliant with the architectural definition, (e) apply a heuristic function for evaluating the configurations, and (f) build the concrete architecture. Next, such operations are described.

Select the candidates: before the execution of the algorithm, candidate components are grouped using the information of the functional part in order to limit the search graph. Each group is related to the operations of a component from the abstract architecture (architectural) and it contains those concrete components which have at least one operation (from provided interface) in common. Thus, graph nodes do not contains more than one component of the same group.

Calculate the configurations: the pseudocode of the developed algorithm is shown in Figure 2. The algorithm starts from an initial node (*source*) which is adjacent to every node created from the components of one group of candidates. Those are the only existing nodes in the graph and the other nodes are created dynamically when a new node is explored (line 25 of Figure 2). Furthermore, new neighbors are created only if the resulting configurations do not exceed the size of the abstract definition (line 22). These optimizations limit the search space of the algorithm and reduce the number of nodes for which the evaluation function $f(x)$ is calculated. In addition, $f(x)$ is used as a reference for managing the priority queue that stores the set of 'open' nodes. From this priority queue, the nodes explored in each iteration are selected (line 10).

The default value for $g(x)$ is 1, since a node differs from its adjacent nodes in the incorporation of one concrete component. However, after the evaluation of the algorithm, there are situations (*e.g.*, when the number of candidate components is very high) in which the establishment of $g(x) = 0$, allows us to obtain architectural solutions in less time. In such cases, the A* algorithm is equivalent to *greedy* search algorithm. This variation means that implementation of the algorithm does not ensure that the resulting solution is the optimal (*i.e.*, the closest to the starting node). Nevertheless, other operations are responsible for checking the algorithm not to add additional components to those defined in the abstract architecture. Moreover, $g(x)$ is configurable through the *Admin* interface of the *Semantic Trader*.

The value of $h'(x)$ represents the distance between the configuration of concrete components (associated with the current node) and the input abstract architectural model (*AAM*). This distance must be 0 (lines 13 and 31) to consider that a configuration is a possible architectural solution, and it is calculated from the semantic information of the components' functional interfaces. This decision ensures (at least) the resolution of valid configuration regarding the functional part, and in less time than if all components' part are evaluated. Nevertheless, when a configuration fulfilling the functional part is found, a full evaluation of the configuration is performed by calculating the distance with respect to the *AAM* and using all the component parts (including extra-functional informa-

```
 1: function aStar(source, AAM)
 2:     openSet ← {source}
 3:     pQueue ← {source}
 4:     closeSet ← ∅
 5:     discardedConfigs ← 0
 6:     notDesiredCC ← ∅
 7:     firstSolution ← false
 8:     bestNode ← ∅
 9:     while openSet ≠ ∅ do
10:         currentNode ← pQueue.poll()
11:         if currentNode.getH() < bestNode.getH() then bestNode ← currentNode
12:         end if
13:         if currentNode.getH() == 0 then
14:             if firstSolution == false then firstSolution ← true
15:             end if
16:             bestNode ← currentNode
17:             if evaluateCAM(currentNode, AAM) == true then return bestNode
18:             else discardedConfigs ← discardedConfigs + 1
19:             end if
20:         else
21:             closeSet.put(currentNode)
22:             if checkCAMSize(currentNode) == true then
23:                 if contains(notDesiredCC, currentNode) == false then
24:                     neighbors = ∅
25:                     neighbors ← createNewAdajectNodes(currentNode)
26:                     for each neighbor in neighbors do
27:                         if contains(closeSet, neighbor) == false then
28:                             if contains(openSet, neighbor) == false then
29:                                 h ← heuristics(neighbor, AAM)
30:                                 newNode ← createNode(neighbor, h)
31:                                 if h == 0 then ... // (lines 14–19)
32:                                 else
33:                                     openSet.put(newNode)
34:                                     pQueue.add(newNode)
35:                                 end if
36:                             end if
37:                         end if
38:                     end for
39:                 end if
40:             end if
41:         end if
42:     end while
43:     return bestNode
44: end function
```

Fig. 2. Search algorithm to find the best configuration

tion). This evaluation also checks: (a) that configurations are closed, *i.e.*, have no components with additional mandatory required interfaces (with regard to the abstract architecture); and (b) that configurations are compliant with the abstract architecture, *i.e.*, functionality is grouped in the components as determined by the architectural definition.

Apply a heuristic function for evaluating the configurations: the method in charge of calculating the scores is *heuristics* (line 29 of Figure 2). This operation involved only the functional part of the components, distinguishing between provided and required interfaces. In order to carry out this process, a 'macro' abstract component, containing all information pertaining to the functional specification of the abstract architecture, is created. Similarly, a particular 'macro' component, which brings together all the functional information of components that are part of the current configuration, is created. In both cases, the union of all provided and required interfaces which are mandatory is produced. These new specifications are compared with the aim of calculating the distance between both definitions (Figure 3).

Matching of provided and required interfaces (MPI and MRI, respectively) is described by a decimal number between 0 and 1, where 0 indicates no match and 1 means a complete match. This value is calculated by dividing the number of matched by total existing operations in the abstract definition. Furthermore, matching of functional part (MF) is calculated as the average of the matching scores from the two types of interfaces, as shown by the following expressions:

$$MF = \frac{MPI + MRI}{2} \qquad MPI = \frac{matchedProvidedOp}{acProvidedOp} \qquad MRI = \frac{matchedRequiredOp}{acRequiredOp}$$

Evaluation function, which represents the distance between an abstract definition and a concrete specification, is calculated from the obtained matching value: $h'(x) = 1 - matching$. Beside this matching value, it is calculated some specific information about which operations (and belonging to which interfaces) of the configuration solve the operations of the functional interfaces described in the abstract architecture. This data is essential for optimizing the performance in the construction of the concrete architecture model, since the relationships between components (and corresponding dependencies between interfaces) are easily calculated from this information.

Moreover, as supplementary information for pairing analysis, other attributes derived from the comparison are calculated: (a) what type of intersection is produced between sets of interfaces, (b) who owns the largest set of provided and required interfaces, (c) the total number of provided and required operations of the configuration and the abstract architecture, (d) the total number of provided and required interfaces. This data is produced as a result of the comparisons made in the *heuristic* method (line 29 of Figure 2).

Closure of configurations and calculation of compliant configurations: both operations are carried out in the *evaluateCAM* function (line 17 of Figure 2). As mentioned above, this method is invoked whenever a configuration is a possible solution, *i.e.*, a configuration whose value of the evaluation function is zero ($h'(x) = 0$). In this function, a new evaluation of matching between the abstract architecture and the current configuration is performed. In contrast to the matching of the *heuristics* method (line 29), the comparison is made for each

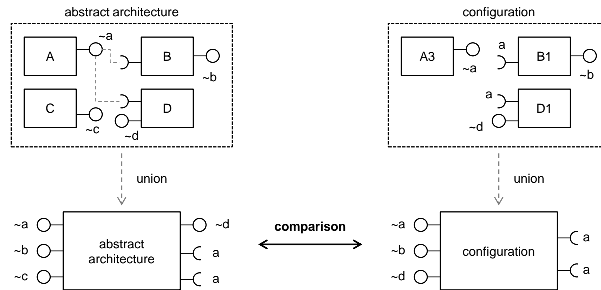

Fig. 3. Comparison between abstract architectures and configurations

component of the configuration independently, instead of performing an overall comparison. Furthermore, this process of matching takes into account all parts of the component specifications. Therefore, four values, corresponding to each of the main parts of the specification of a component, are calculated:

— *Functional information*: scoring process results in the MF value described for the *heuristics* method.
— *Extra-function information*: the total matching value is the average of the matches of dependencies and properties parts. Additional information is also calculated, as the type of intersection between the sets compared or their relative size. Extra-functional information is divided in: a) *Properties*: firstly, the matching operation checks which properties of the abstract component are fulfilled in the concrete component. Secondly, the matching value is calculated as a weighted sum of the three categories of existing properties (high, normal or low priority). In this sense, the matching of properties with high priority involves higher matching scores than meeting properties with a normal (or low) priority; b) *Dependencies*: for calculating this value of matching, it is necessary to take into account the type of intersection between the sets of dependencies. If there is no intersection, matching is 0.0, provided that both sets are not empty (in which case matching is 1.0). If there is intersection, there may be three possibilities: (1) that all the dependencies of the concrete component (DCC) are within the set of abstract component dependencies (DAC), (2) that DAC is within DCC, and (3) that no set is within the other. Next, the expressions to calculate the value of matching (m), depending on the three possibilities and taking into account the number of dependencies in the intersection ($matchedDep$), are shown:

$$(1)\ m = \frac{matchedDep}{card(DAC)} \qquad (2)\ m = \frac{matchedDep}{card(DCC)} \qquad (3)\ m = \frac{matchedDep}{card(DAC) + card(DCC)}$$

— *Packaging information*: the total matching value is the average of the matches of implementation and location parts. Additional information as the type of intersection between the sets or their relative size is also calculated.
— *Marketing information*: calculated scores represents if the components are developed by the same entity and if same contact people are associated.

Each matching value is described by a decimal number between 0 and 1 (where 0 indicates no match and 1 means that the matching is complete). Thus, total matching between two components is calculated from the matching of the functional part (MF), matching of the extra-functional part (MEF), matching of packaging information (MP), and matching of marketing information (MM). Figure 4 shows a graphical representation of calculated matching scores for three concrete components. On the one hand, we can see that the component which best meets the abstract definition is $CC3$. On the other hand, the representation of the matching score of component $CC1$ has a larger area (determined by the four points of the four components' parts) than the component $CC2$. However, depending on the importance we attach to each part, it can be considered as the component $CC2$ meets better the abstract specification.

In this sense, matching score between two components is calculated from the following expression: $matching = MF * factorMF + MEF * factorMEF + MP * factorMP + MM * factorMM$. By default, the values for *factorMF*, *factorMEF*, *factorMP* and *factorMM* are 0.8, 0.15, 0.025 and 0.025, respectively. Therefore, we give more weight to functional and extra-functional properties. Nevertheless, these weight can be modified at run-time using the *Admin* interface of the trading service (if the condition that the sum of the factors is equal to 1 is satisfied). As a consequence, it is possible to vary the weight given to each of the components' parts for comparison operations.

Resuming the execution of *evaluateCAM* method, the closure operation verifies there are no additional dependencies (mandatory required interfaces) in concrete components with respect to the abstract definition. With the aim of checking the compliance with the architecture, it is verified that the configuration is made up of the same number of components as the abstract definition. When the trading process is set up, the minimum distances to consider that a configuration meets architecture can be established. In addition, this operations of the *Admin* interface can be used at run-time to modify the execution policies. For example, it is possible to specify that a ratio of 0.95 for extra-functional properties must be accomplished, thus determining that matching score of that specific part cannot be less than this value. Those configurations that are not closed or do not comply with the architecture, are discarded, continuing the search algorithm until a valid solution is found.

There is another remarkable feature implemented in this process. A maximum value of time that should not be exceeded to obtain a valid solution can be defined. Thus, when this limit is exceeded, the search algorithm stops, although it has failed to reach the final solution. Nevertheless, although several solutions have been discarded, it is highly probable that other configurations have been found with a value of $h'(x)$ equal to 0.0. Such configurations are valid solutions in functional terms. Therefore, whenever a configuration of this type is found, a reference to the corresponding node is saved, since it is the best node found up to that point (line 16 of the algorithm shown in Figure 2) In the case that the

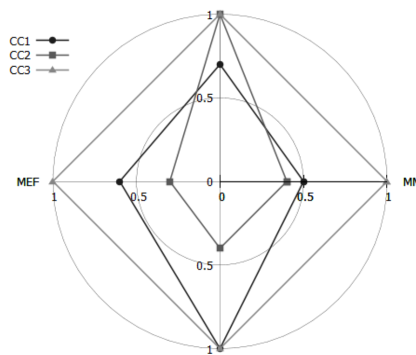

Fig. 4. Example of matching scores

algorithm must terminate without finding the final solution, this configuration will be returned by the algorithm, to ensure the resolution of the architectural definition (at least at the functional level).

Build the concrete architecture: once the final solution has been obtained, the semantic trading service constructs the concrete architectural model associated with the configuration found by the A* search algorithm.

3 Related Work

In the construction of software architectures, selection and evaluation processes are considered as key operations [14]. An example of work in which these processes are addressed is the Off-The-Shelf Option (OTSO) [12]. In such approach, a hierarchical evaluation criteria analyzes the characteristics of the components based on other factors such as organizational infrastructure or the availability of libraries. In [9], DesCOTS system proposes a methodology based on a quality model which divides the characteristics of the components for their evaluation.

The work presented in [17] evaluates the components and establishes a ranking in terms of performance and according to multiple criteria. In [7], authors perform a management of dependencies between components using goal-oriented models as the basis for component selection. A proposal for selecting COTS components in large repositories is described in [2]. The approach makes use of the 'integrator' concept instead of mediation or trading services. In contrast to our proposal, the approaches mentioned above do not support component selection or calculation of configurations at run-time.

The trading described in [10] is the basis of the work. It presents a mediation process for managing COTS components and building configurations at *design-time*. In contrast, our approach is intended to build architectures at *run-time* based on a semantic matching of components. Algorithms based on heuristic functions are a suitable option for the exploration and evaluation of possible solutions [13]. In addition, other type of algorithms, such as exhaustive search algorithm for building configurations of components [4], results in exponential execution orders because all the nodes in the search space must be evaluated.

4 Conclusion and future work

We presented an approach for matching component specifications at run-time. This operation is part of a trading service in charge of dynamically building architectures. This process is responsible for calculating the best architectural solutions starting from their corresponding architectural definitions. In order to address this resolution, matching operations are performed to compare components and obtain scores describing the distance between: (a) an input architectural definition and (b) each of component configurations which are possible solutions of the architecture. Furthermore, these scores are calculated applying an evaluation function that makes use of the syntactic and semantic information

described the component specifications. This evaluation function is utilized in an A* algorithm as the heuristic to find the best configuration.

There are identified some lines as future work. We plan to extend the matching information calculated from the comparison of two components or architectures. In addition, we plan to evaluate alternative search algorithms for building architectures at run-time. Furthermore, we plan to improve the performance of the proposed algorithm, for example, parallelizing part of the execution. Moreover, we intend to develop some tools for querying and managing the information of component specifications and their comparisons.

Acknowledgments. This work was funded by the Spanish Ministry MINECO under Project TIN2013-41576-R, the MECD under a FPU grant (AP2010-3259), and the Andalusian Government (Spain) under Project P10-TIC-6114.

References

1. Carney, D., Leng, F.: What Do You Mean by COTS? Finally, a Useful Answer. IEEE Software, 17(2), 83–86 (2000)
2. Clark, J., *et al.*: Selecting components in large COTS repositories. Journal of Systems and Software, 73(2), 323–331 (2004)
3. Criado, J., *et al.*: Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces. SPE Jour., Elsevier (2014)
4. Criado, J., Iribarne, L., Padilla, N.: Resolving Platform Specific Models at runtime using an MDE-based Trading approach. LNCS 8186, pp. 274–283 (2013)
5. Crnkovic, I.: Component-based Software Engineering – New Challenges in Software Development. Software Focus, 2(4), 127–133 (2001)
6. Crnkovic, I.: Component-based Software Engineering for Embedded Systems. 27th ICSE, pp. 712–713. ACM (2005)
7. Franch, X., Maiden, N.A.: Modelling component dependencies to inform their selection. COTS-Based Software Systems, pp. 81–91. Springer (2003)
8. Graham, S., *et al.*: Building Web services with Java: making sense of XML, SOAP, WSDL, and UDDI. SAMS publishing (2004)
9. Grau, G., Carvallo, J.P., Franch, X., Quer, C: DesCOTS: a software system for selecting COTS components. 30th Euromicro Conf., pp. 118–126. IEEE (2004)
10. Iribarne, L., Troya, J.M., Vallecillo, A.: A trading service for COTS components. The Computer Journal, 47(3), 342–357 (2004)
11. ISO/IEC 13235-1, ITU-T X.950. Information Technology – Open Distributed Processing – Trading function: Specification (1998)
12. Kontio, J., Caldiera, G., Basili, V.R.: Defining factors, goals and criteria for reusable component evaluation. In: CASCON'96, pp. 21–32. IBM Press (1996)
13. Korf, R.E.: Real-time heuristic search. Artificial Int., 42(2), 189–211 (1990)
14. Mohamed, A., Ruhe, G., Eberlein, A.: COTS selection: past, present, and future. Engineering of Computer-Based Systems, pp. 103–114. IEEE (2007)
15. Mishra D, Mishra A.: Distributed information system development: review of some management issues. OTM 2009 Workshops, pp. 282–291 (2009)
16. Mishra D, Alok M. Research trends in management issues of global software development: evaluating the past to envision the future. J. GITM 14(4), 48–69 (2011)
17. Shyur, H.J.: COTS evaluation using modified TOPSIS and ANP. Applied Mathematics and Computation, 177(1), 251–259 (2006)