Feature-Based Discovery of Services with Adaptable Behaviour

J. Antonio Martín and E. Pimentel
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga
Málaga, Spain

e-mail: {jamartin, ernesto}@lcc.uma.es

Abstract—Service discovery is one of the corner stones of Service-Oriented Architectures. It aims at finding the set of services which comply with a query in a scalable way with regard to the number of services in the registry. Current approaches to service discovery are based on registries (like UDDI) which provide efficient search algorithms over the interfaces (described in WSDL) and metadata of the services, or are focussed on ontology matching mechanisms over semantic descriptions of the services (e.g., OWL-S). However, these approaches rarely support searching over stateful services (such as those described as Windows Workflows or BPEL processes) based on behavioural queries or, when they do, they perform the behavioural analysis on a second step, due to its computational complexity, and then they discard the services which present deadlocks or livelocks. In this work, we introduce a new search tree for the discovery of stateful services which can be adapted to fulfil behavioural queries. The behaviour and adaptability of the services is analysed upfront during the discovery process. Thanks to Software Adaptation, our approach returns both services which perfectly match the query and those which can be adapted to it. This adaptation overcomes incompatibilities in interface and behaviour, which could lead to deadlock or livelock situations, hence our approach returns more services than traditional techniques. Moreover, behavioral adaptation abstracts services into smaller models which enable service discovery in polynomial time.

Keywords—service discovery; behavioural adaptation; search tree; Web service; SOA

I. INTRODUCTION

Service discovery aims at finding services which match a query with the purpose of directly accessing the service or including it into an orchestration. However, Web Services (WS) are not always compatible and, even though they might present the functionality and QoS expected by the query, they cannot be directly used because of these incompatibilities. These incompatibilities can be present in their interfaces but also in their behaviours. The latter applies to stateful services (such as those described as BPEL processes [1] or Windows Workflows [2]) where any mismatch in the sequence of the messages exchanged may lead the composition to a deadlock situation. For instance, a missing operation in a service, a mismatch in the operation name or arguments, or an unexpected sequence of messages makes impossible the correct termination of the services involved.

Preprint version. This article was published in Proc. of ECOWS’10, doi:10.1109/ECOWS.2010.22

Figure 1. Overview of the discovery of adaptable services

Service Discovery (both at run time and design time) has to face this issue. We assume that, similarly to behavioural services, discovery queries have interface and behaviour. We call these behavioural queries. Then, the services to be discovered must comply with the interface and behaviour required by the discovery query, otherwise incompatibilities might lead the whole system to deadlock situations where none of the services can reach a stable state.

Software adaptation [3], [4], [5] is a sound solution which enables Web Services to interoperate despite their initial incompatibilities. This adaptation is achieved by deploying an adaptor, either as a set of wrappers or as a centric orchestrator, which is in charge of receiving, translating and rearranging the messages in the way expected by the destination service.

Adaptors can be either generated beforehand and stored for their later use, or created automatically [6], [7] at run time. Adaptation is based on common features between the expected and the provided functionality, i.e., the adaptor can change the signature and behaviour of the service but it cannot change the underlying functionality and semantics of the service. Therefore, adaptation is not possible in the cases where there is not such a mapping between expected and provided features (e.g., no adaptor can make a booking out of a weather service).

Contribution: In this work, we propose to apply software adaptation techniques to service discovery. The goal is to discover those services in the registry whose behaviours can be adapted to the query (see Fig. 1). Once discovered, these services can be adapted and composed
using previous work [3]. Therefore, we focus on defining the conditions for adaptability and designing a scalable search tree over services depending on those conditions. The benefits of using software adaptation in the discovery of behavioural (i.e., stateful) services are threefold: i) the behaviour of the services can be abstracted to a reduced set of features required for the adaptation; ii) feature-oriented representation of the services decreases the computational complexity of the discovery process; and iii) discovery of adaptable services obtain more results as it gathers services that perfectly match the query and those which can be adapted to fulfill the interface and behaviour required by the query.

We will exemplify the model and algorithms involved with a query (Fig. 2) and two services inspired in some operations of the Picasa Web feed (Fig. 3) and Flickr SOAP API (Fig. 5). We use these services to illustrate feature-based adaptation in Section II. The adaptation requirements presented in the previous section are used in Section III to reduce the services to the dependencies among the their provided and required features. Section IV introduces the search tree we developed to support the discovery process based on feature-based service abstraction. We discuss some related work (Section V) and we conclude with some final remarks in Section VI.

II. FEATURE-BASED BEHAVIOURAL ADAPTATION

Behavioural Adaptation [4] is based on the special role of the adaptor as man-in-the-middle of the communication which intercepts the requests of the services, recompose those requests as expected by the destination service(s), and replies with the appropriate message at the appropriate time.

Message names and structures are altered as needed by the adaptor (the adaptor merges, splits and transforms messages to overcome incomparabilities), hence the restrictions to the adaptation are service controllability, the data received and sent by the adaptor, non functional properties and the side-effects of the operations. Controllability will be covered in Section III and we handle data, non functional properties and side-effects as features of the operations in the service behaviour. These features can be semantic annotations on the behaviour of the service or obtained from the arguments in the service signature, for example. Features can be either provided or required by parts of the service behaviour.

The adaptor can only provide features which have been previously offered to it. For example, the adaptor cannot create new data so any argument sent by the adaptor must be previously received. Following this reasoning, the only feature-wise restriction to the adaptability of the composition of several services is that some service is able to provide a feature before another service requires it. In the case of data, these might be reused in several iterations, therefore the adaptor can provide several times an already received feature, e.g., the correlation set parameter in BPEL, which must be resent in every message within the same session.

We model service behaviour with Labelled Transition Systems (LTS, [8]) defined as follows:

Definition 1 (Service Behaviour): A service behaviour is formally defined as a LTS \( (\Sigma, O, s_0, F, t) \) where: \( \Sigma \) is an alphabet which corresponds to the set of labels associated to transitions, \( O \) is a set of states, \( s_0 \in O \) is the initial state, \( F \subseteq O \) are final (or stable) states, and \( t \subseteq (O \times \Sigma \times O) \) are the transitions.

The labels in \( \Sigma \) are either internal transitions (\( \tau \)) or communication transitions which start with the operation name followed by an ‘!’ or ‘?’ if they are output or input actions, respectively. Labels also contain the list of features, which are provided features in output actions and required features in input actions. Labels are expressed with the following notation: \( \text{operation}(\![!/?]\{, \text{feature} \})^* \).

This formal model of service behaviour has been chosen because it is simple, graphical, and it can be easily derived from existing implementation languages (see for instance [9], [10], [11], [12] where such abstractions for Web services were used for verification, composition or adaptation purposes; and [13] for a description of the operational semantics behind the composition of this kind of services).

Discovery queries represent the complementary behaviour of what is expected from the service to be discovered. Therefore, queries are modelled as service behaviours with complementary communication actions.

Example 1: Fig. 2(a) represents the behaviour and interface required by the discovery query. The query contains an initialisation message with a feature \( K \) (an authentication token), then it requests to login with \( \text{Usr} \) and \( \text{Pass} \), it sends a message asking for pictures and finally expects to receive one or several pictures with their \( \text{Url} \) and information (\( I \)). The Picasa Web in Fig. 3(a) provides similar functionality to the one requested by the query. However, the Picasa service can provide pictures to anonymous and authenticated users. In the latter case, it

http://code.google.com/apis/picasaweb/docs/2.0/reference.html
http://www.flickr.com/services/api/
additionally provides the comments of the pictures (Com) and it requires a Code to end the session.

The adaptor represented in Fig. 4 would make the behaviour of the query (Fig. 2(a)) and the behaviour of Picasa Web (Fig. 3(a)) to cooperate properly in spite of the incompatibilities in the number and name of the messages, and their arguments. For instance, the login and getFeed request in the query should correspond to the getFeed operation in Picasa Web, therefore two messages are transformed into a single request, feature Pass is not used and none of the operation names are compatible (i.e., the same between services). Communication actions in the behaviour of the adaptor have been prefixed with ‘q:’ (the query) and ‘p:’ (Picasa Web) depending on the service that must synchronise with the adaptor. In this example, features represent the arguments sent and received by the service and the adaptor. When the adaptor needs to send a message to a service with some certain features (e.g., pushPic?Url,I), the adaptor should have received such features from previously processed messages (e.g., getFeed?Usr,Url,I).

There are several approaches to generate adaptors either automatically [6] or based on adaptation contracts that must be previously designed [3].

Henceforth, discovery queries are considered to be equivalent to behavioural services, therefore the discovery process will consist in checking whether the query and each of the services in the registry are adaptable or not.

III. ABSTRACTING ADAPTABLE BEHAVIOUR

A. AD trees

Message names are irrelevant in terms of adaptability, therefore we can abstract the behaviour of the services and reduce them to the possible interactions in which the features are exchanged. We will refer to this control-flow abstraction as Adaptation Dependency Tree or AD tree. An AD tree is an abstraction of all the possible traversals of the behaviour of the service. AD trees will serve us to analyse the controllability of the query and the services based on their features.

Definition 2 (Adaptation Dependency Tree): An Adaptation Dependency Tree (AD tree) is a tree with OR nodes, AND nodes and END nodes where edges are labelled with a (possibly empty) set of features. Sets of features are prefixed by ‘?’ or ‘!’ depending on whether these features are required or provided, respectively.

Example 2: Fig. 2, Fig. 3 and Fig. 5 show the behaviour and AD tree abstractions of the query, Picasa Web and Flickr services, respectively. Dashed transitions in the AD tree represent different interleaving cases of the operations covered by solid transitions, therefore these parts have been omitted for the sake of clarity. There are several differences between Flickr and Picasa Web. Flickr uses a token K (the API key) to associate every received message with a session, and it uses another token called frob (F) during the authentication. Additionally, instead of returning

Trees are drawn as directed-acyclic graphs with starting and ending nodes for simplicity reasons.
the Url of the pictures directly, it is required to request the collection of the Usr (C), then receive the information of that collection (Ti, Desc and P), and finally use P to obtain the Url of the picture.

In AD trees, there are OR nodes and AND nodes which can have several children. On one hand, OR nodes represent that the choice of which branch to follow depends on external communications, i.e., any external service or the adaptor can decide by sending a particular message which branch is going to offer the current service (e.g., a PICK activity in BPEL). On the other hand, AND nodes are internal choices of the service, therefore the composition with this service must deal with every branch outgoing from these nodes (e.g., a IF activity in BPEL).

Edges are labeled according to whether the service provides/sends (prefixed with exclamation marks) or requires/receives (prefixed with question marks) certain features. Whether these features are given in a single or several messages is irrelevant for the AD tree, therefore sequences of edges with the same type are collapsed in a single edge. For instance, the features provided by initialise and login in the query (Fig. 2(a)) are collapsed in a single edge in its AD tree (!K,Usr,Pass in Fig. 2(b)).

Adaptors are able to reuse previously received features, because of this, once a given feature occurs in an edge, that same feature does not appear in any descendant edge.

In order to check the adaptability between two given AD trees we must start at the root of both AD trees and traverse them while matching the branches as follows:

1) If it is an AND node, there must be a matching branch in the other AD tree for every branch outgoing from this AND node.
2) If it is an OR node, we can continue with any (or several) of the outgoing branches.
3) If it is a ‘?’ edge, every feature in that edge must be present in ancestor ‘!’ edges of the other AD tree.
4) Edges with provided features (i.e., ‘!’ edges) can be ignored in the matching unless required by ‘?’ edges in the other AD tree (step 3).
5) Un-labelled nodes are skipped and END nodes must match with END nodes in the other AD tree.

Example 3: The AD tree representing the query (Fig. 2(b)) matches with both Picasa Web (Fig. 3(b)) and Flickr (Fig. 5(b)). Bold edges with empty arrowheads in the behaviour and AD trees of the services represent the branches that fulfilled the query. Let us note that our approach automatically recognised and overcame the dependencies in Flickr (F, C and P) to match the query. In the particular case of Flickr (blue-bold transitions with empty arrowheads in Fig. 5), the adaptor must first get the collections of pictures, then get the information about those collections and finally obtain the URLs of the photos in each collection.

Using AD trees for comparing queries and services is less demanding in computational terms than comparing behaviours. The reason is that we compare trees (limited in depth by the number of features) instead of graphs, which can entail exponential complexity.

B. Feature-Dependency Rules

Comparing AD trees requires still some complexity due to the nature of AND nodes (where the comparison must be scattered among several branches, step 1) and OR nodes (where the comparison is based on a try-and-error approach in every branch, step 2). Additionally, AD trees avoid loops but they suffer the interleaving of operations during the execution of the service. For example, Fig. 5(b) has several omitted parts (dashed edges) which contain other interleaving cases of the edges represented in the solid branches. This is the reason we abstract AD trees and reduce them further in Feature-Dependency Rules.
where A with similar requirements we can discard the rules which are denoted by:
\[ f \leftarrow \{f_1', \ldots, f_n'\} \ | \ \{f_1', \ldots, f_m'\} \]
where \( f_i', f_j' \in F, i \in [1, n], j \in [1, m] \) and F is the set of features.

A FD rule \( f \leftarrow F' \ | \ F'' \) represents that a given service provides feature f if it is given the required features \( F' \). In addition, features \( F'' \) are needed for that service to reach a final/stable state after f is provided. These are ending features. Every FD rule preserves that \( F' \cap F'' = \emptyset \).

The intuition behind FD rules is to express which features are required in a single session with a particular service to obtain a given feature and complete the session. Behavioural adaptors support reordering [14], therefore the adaptation is possible if the adapted service corresponding to the rule is provided with the required features in any order (required and ending features are sets), then we obtain the requested feature f and, finally, the ending features must be given in any order as well for the adapted service to end up properly.

Abstracting service behaviour into sets of FD rules we lose the correlation among the behaviour of the service and the different rules, that is, it might happen that even though a service complies with two rules, it cannot offer both during the same session. In those cases, we assume that we can establish several concurrent sessions with the same service (e.g., it is possible to have several concurrent instances of the same BPEL process [1]), and that service does not qualify to provide that feature.

FD rules are obtained from the AD tree of a given service. While traversing an edge in the AD tree, every provided feature in that edge generates a new FD rule, all the features which were needed to reach that edge are the required features of the rule and, from that node on, every feature needed to reach a final state is included as an ending feature. If the same feature can be obtained in different paths with OR nodes, several FD rules are generated to provide the same feature. In addition, AND nodes must be considered in an special way:

- If a feature is not provided in every branch of an AND node, no FD rule is generated for that feature after the AND node. This means that, if the service can internally decide whether to provide or not a feature, that service does not qualify to provide that feature.
- If the feature is provided in every branch of an AND node, all the required and ending features gathered in those branches are joined in the same rule. That means that the rule requires everything that might be needed regardless the internal choices of the service.

If there are two FD rules which provide the same feature with similar requirements we can discard the rules which demand more features. This is formalised in [Definition 4].

**Definition 4:** (FD rule partial order) Two FD rules \( a = f \leftarrow A' \ | \ A' \cup C \) and \( b = f \leftarrow B' \cup C \ | \ B' \) are related by a partial order \( (a \preceq b) \) if, and only if, \( A' \subseteq B' \) and \( A' \subseteq B' \) where \( A', A'', B', B'' \) and \( C \) are sets of features.

The intuition behind this partial order is that a rule requires less than another if it has less required features, less ending features or some required features are placed as ending features. The first two conditions are straightforward and the third boils down to say that the rule requires the same set of features but it requires them later, therefore the rule is less restrictive.

From a generated set of rules R, we keep only the minimal rules according to this partial order (i.e., \( \{ r \in R \ | \ \exists r' \in R, r' \preceq r \} \)). Therefore, minimal rules subsume all the other more demanding rules.

A query represented by an AD tree can be transformed into a set of FD rules. This is done in the following steps: i) we replace the features in the AD tree of the query with their complements; ii) we generate all the possible combinations of AD trees that would be created by replacing every OR node in the AD tree of the query with one of its branches; iii) we convert each of these AD trees into FD rules following the algorithm described previously. In this way, we obtain a set of sets of FD rules. The upper bound for the number of generated FD rules is exponential with regard to the number of different features in the query but it is independent of the behaviour of the query and it is reduced by the partial order between FD rules. Any service whose FD rules subsume (with regard to ‘\( \preceq \)’) all the FD rules of one of these sets of FD rules can be adapted to fulfil the query.

**Example 4:** The FD rules of the query (Fig. 2(b)) are:

\[
I \leftarrow \{K, Usr, Pass\} \ | \ \emptyset \quad (1)
\]
\[
Url \leftarrow \{K, Usr, Pass\} \ | \ \emptyset \quad (2)
\]

by abuse of notation we can omit empty sets and we can collapse several rules into a single expression. For instance, rules (1) and (2) can be represented as:

\[
I, Url \leftarrow \{K, Usr, Pass\} \ | \ \emptyset \quad (3)
\]

The FD rules of Picasa Web (Fig. 3(b)) are:

\[
I \leftarrow \{Usr\} \quad (3)
\]
\[
Url \leftarrow \{Usr\} \quad (4)
\]
\[
Com \leftarrow \{Usr, Pass\} \ | \ \{Code\} \quad (5)
\]

rules (3) and (4) subsumed rules \( I, Url \leftarrow \{Usr, Pass\} \ | \ \{Code\} \) which correspond to the authenticated part of Picasa Web, where feature Code was required to finish the session. The FD rules of Flickr (Fig. 3(b)) are:

\[
F, Tok, Perm, C, Ti, Desc, P \leftarrow \{K, Usr, Pass\} \ | \ \emptyset \quad (6)
\]
\[
I \leftarrow \{K, Usr, Pass\} \quad (7)
\]
\[
Url \leftarrow \{K, Usr, Pass\} \quad (8)
\]

Rule (1) is subsumed by the rules (3) and (7). Similarly, rule (2) is subsumed by the rules (4) and (8). As expected, both services satisfy the query.

FD rules are the key to process discovery queries, therefore we focus on these to develop a search tree, called FD tree, which performs well in finding those services which comply with a certain FD rule.
IV. SEARCH TREE OF ADAPTABLE WEB SERVICES

FD trees represent the registry of services. FD trees return those services which comply with a given FD rule. A discovery in the registry corresponds to several traversals of the FD tree, one per FD rule in the query.

We will describe FD trees using a generic FD rule \( f \leftarrow F_f \mid F_e \) as the query. A FD tree consists in a first level of edges which are labeled with the feature that is requested \((f)\). From that point on, every node in the tree has two sets of services: one set contains the services that are able to provide the feature in that point, and the other set contains the services that are able to finish in that point. Every edge will be labeled with a different feature, first the required features \((F_f)\) and then the ending features \((F_e)\). In this way, in a single trace throughout the tree, the search returns all those services which comply with a FD rule.

The search algorithm (Algorithm 1) processes each FD rule in the query (lines 2-23), it traverses the FD tree using the required features first (lines 7-11) and ending features later (lines 12-15) while collecting the desired services.

Considering set structures that perform membership queries in logarithmic time, a FD rule query has a complexity in time of \( O(S \cdot \log(S) \cdot F) \), where \( S \) is the number of services in the registry and \( F \) is the number of possible different features. Therefore, the discovery of all the services which can be adapted to a given behavioural query has a time complexity of \( O(Q \cdot S \cdot \log(S) \cdot F) \), being \( Q \) the number of FD rules in the query.

Example 5: The FD tree in Fig. 6 contains the FD rules that represent Picasa Web and Flickr. Transitions outgoing from dashed nodes have been omitted for the sake of clarity as they represent different interleaving of sequences expressed in other paths. As an example, the Picasa rule \( \text{Com} \leftarrow \{ \text{Usr, Pass} \} \mid \{ \text{Code} \} \) is included as a first transition with \( \text{Com} \), two transitions with \( \text{Usr} \) and \( \text{Pass} \), then the service can provide \( \text{Com} \) (so it is placed in the left-hand side set of services) and finally Picasa Web can end after receiving \( \text{Code} \).

It is worth mentioning that the interleaving in FD trees, which is the reason for the FD tree having several superfluous nodes, can be avoided by imposing an arbitrary order among features. In order to favour discovery requests, features should be sorted according to their frequency as required features and ending features in service behaviours.

Example 6: Fig. 7 displays the FD tree corresponding to the one in Fig. 6 but with an additional service (a) and enforcing an order among features (\( \text{Usr} < \text{Pass} < \text{K} < \text{Code} < \ldots \)). Let us note that this enforced order must be respected by the loops in lines 7 and 12 of the discovery algorithm (Algorithm 1).

An application of FD trees due to their efficient search algorithm and their support of behavioural adaptation is compositional discovery, where the process searches for possible compositions of services that, together, achieve to fulfill the query. Compositional discovery allows to obtain the needed features from different services and, in addition, we can extend the results with services that are further down in the tree than what is allowed by the features (required and ending) of the original query. In this way, the additional features needed for reaching these services generate new discovery queries whose required and ending features are the same as the original query. Once the

Algorithm 1 discovery

Processes a discovery query considering feature-based behavioural adaptation

inputs: An FD tree and a query consisting in a set of FD rules

output: Set of services which can be adapted to fulfil the query

1: \( \text{sol} = \emptyset \)
2: for all \( f \leftarrow \{ f_1, \ldots, f_n \} \in \text{query} \) do
3: \( \text{match} = \emptyset \); \( \text{part} = \emptyset \); \( \text{end} = \emptyset \)
4: \( \text{node} = \text{tree.root getChild}(f) \)
5: \( \text{match.addAll(node.endingServices)} \) \{Ending services are feasible solutions\}
6: \( \text{part.addAll(node.providedServices)} \) \{Services that must be checked for their ending features\}
7: for all \( f_i \in f_1, \ldots, f_n \) do \{Follow required features\}
8: \( \text{node} = \text{node.getChild}(f_i) \)
9: \( \text{match.addAll(node.endingServices)} \)
10: \( \text{part.addAll(node.providedServices)} \)
11: end for
12: for all \( f_i \in f_1, \ldots, f_n \) do \{Check ending features\}
13: \( \text{node} = \text{node.getChild}(f_i) \)
14: \( \text{end.addAll(node.endingServices)} \)
15: end for
16: \( \text{match.addAll(part \cap end)} \)
17: if \( \text{init} \) then
18: \( \text{sol} = \text{match} \)
19: \( \text{init} = \perp \)
20: else
21: \( \text{sol} = \text{sol} \cap \text{match} \) \{Return services which satisfy every rule in the query\}
22: end if
23: end for
24: \( \text{return sol} \)
some transitions are missing in the query. We go beyond such discrepancies since our approach supports adaptation, therefore we analyse if such behavioural discrepancies can be automatically adapted. In addition, their performance evaluation [15] states that the behavioural comparison among services and the query amounts for most of the computation time even after the previous filtering stages whereas our approach, centred in behavioural adaptation, keeps a polynomial complexity. The goal of our work is to filter the services in the registry by whether they can be adapted to fulfil the query or not, so our work could be used as an initial filter in their architecture. In this case, their functions to evaluate the interface and behavioural distances should be customised to consider the adaptation effort of the service instead of just counting missing edges.

A similar approach is presented by Benigni et al. [18] where the discovery process is divided in two steps: functional analysis (where they check I/O dependencies among services using OWL-S ontologies) and behavioural analysis, where they verify deadlock freedom through Petri Nets obtained from their OWL-S descriptions. Their work also includes the automatic discovery of minimal service compositions which fulfil the query. Like the previous related work, they do expensive behavioural analysis on a second step, therefore behavioural discovery does not have the first-citizen status as it is given by our work. In addition, they overcome certain incompatibilities due to the ontology relationships expressed in OWL-S but they do not consider the presence of adaptors in the composition, therefore they are not able to achieve the level of adaptation supported by our approach.

Both previous approaches focussed on comparing one-by-one the query and the services (or composition of services). Our work, however, is centred in the creation of search structures to enable scalable services discovery where the returned services can be adapted automatically to fulfil the query. Our search tree for service discovery was inspired by graph indexes commonly used in the area of chemistry to find molecules among a database of compounds. This area puts a lot of emphasis in the scalability of the algorithms and some of the proposals [19], [20] are based on graph sub-isomorphism queries and different notions of distance between the graph of the query and the sub-isomorphism found in the compound. A lot of this knowledge could be reused for the discovery of behavioural services. However, chemistry related indexes are usually too optimised to non-directed and non-labelled graphs to be easily customised to feature-enabled discovery of adaptable services.

To our knowledge, there is no other service discovery approach enhanced with behavioural adaptation and supported with a search tree. In most other work, incompatibilities among services are restricted to missing transitions or ontology matching and analysis, and they do not take advantage of all the efforts made in Software Adaptation [4], [5].

VI. CONCLUSIONS AND FUTURE WORK

In this work, we have presented a search tree over an abstraction of behavioural services which is able to discover
services that can be adapted to fulfil the query. Software adaptation enhances service discovery by allowing to find both services which perfectly match the query and services that could be automatically adapted to the query. Therefore, this approach returns more results than traditional service discovery techniques without adaptation.

The discovery of adaptable services is based on features provided and required by the services and the query. The feature-based abstraction of stateful services presented in this work resulted in smaller and more efficient representation of services. These abstract representations of adaptable services are gathered into a search tree structure to support efficient discovery queries. Queries executed on this structure have a computational complexity of \( O(Q \cdot S \cdot \log(S) \cdot F) \) where \( S \) is the number of services in the registry, \( F \) is the number of possible different features and \( Q \) is the number of FD rules in the query. The services discovered through this search tree are guaranteed to be adaptable using previous work for the generation of behavioural service adaptors [3]. We consider that this low complexity, the additional results obtained because of adaptation, and the inclusion of behavioural concerns in the discovery makes our approach well suited for the scalable discovery of stateful services.

Regarding future work, we plan to handle differently features that should not be reused by the adaptor or explicitly avoided by it. These features could be side effects, such as payments or removals, which require further considerations than those proposed in this paper. This could be done by enforcing LTL properties over the features of the services as a second stage of the discovery. Finally, we plan to further work on compositional discovery and we have to evaluate the memory requirements of the search trees presented in this work and the complexity of creation, updates and removals of services from the search tree.

**ACKNOWLEDGMENT**

This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Education and Science (MEC), FEDER and project P06-TIC-02250 funded by the Andalusian local Government.

**REFERENCES**


