JAMEJAM: A Framework for Automating the Service Discovery Process

Islam Elgedawy
Computer Engineering Department, Middle East Technical University, Northern Cyprus Campus, Guzelyurt, Mersin 10, Turkey.

* Corresponding author: Tel.: +90-392-6612963. email: Elgedawy@metu.edu.tr
Manuscript submitted January 25 2016; accepted April 25, 2016.
doi: 10.17706/jsw.11.7.646-655

Abstract: The service discovery problem is not trivial, as it requires solutions for many complex sub-problems such as service semantic description, service identification, service composition, service selection, service evaluation, service adaptation and presentation. Currently, companies manually construct their discovery processes in an ad-hoc tightly-coupled manner using different platform-services that separately handle the identification, composition, selection, evaluation, adaptation and presentation sub-problems. However, when users' requirements change, the already constructed discovery process needs to be manually reconstructed and reevaluated again. This creates a need for an automated approach that allows different users to dynamically construct their discovery processes on the fly. Therefore, we propose JAMEJAM, a framework for service discovery automation. It enables users to create their customizable discovery processes on demand as an executable BPEL process that describes the required matching aspects, matching schemes and matching policies. JAMEJAM realizes such process by dynamically searching for the suitable platform-services in a context-sensitive manner using different types of knowledge (e.g., aspects, services, and matching schemes knowledge), captured via different software ontologies. Experimental results show that JAMEJAM increases the accuracy and the adaptability of the service discovery process.

Key words: JAMEJAM, discovery analytics, service discovery, software ontology.

1. Introduction

According to the internet of services vision, users (i.e., people, businesses, and systems) should allocate and consume the required computing services via the Web in a context-aware seamless transparent manner, according to predefined Service Level Agreements (SLAs). Hence, services need to find other services in the Web without a priori knowledge of their existences. This is known as the service discovery problem. Such discovery problem is not trivial, as it requires solutions for many complex problems such as service semantic description, service identification, service composition, service selection, service evaluation, service adaptation and presentation [1], [2]. The output of a discovery process is a list of atomic and/or composite services that fulfill users' requirements. A discovery service is the service that implements and executes the service discovery process. We believe the service discovery process should be divided into the following main sequential stages: **1) Service Description:** Service providers should provide services' descriptions in a machine-understandable format to enable discovery services to understand the services' semantics and use them to accurately identify the services suitable for users' requirements. Service

description must include different service aspects. A service aspect is a specific view, an interpretation, a facet, a distinct feature about the service. An aspect could be emergent (such as behavior, security, performance), or non-emergent (such as interfaces, architecture patterns, business scopes). **2) Service Matching and Identification:** Users' functional and nonfunctional requirements are given as queries to the identification and matching platform-service. The output of this stage is a list of candidates. Such list is obtained by applying different service matching and identification approaches that examine different aspects of the service. Every aspect is examined via a specific matching scheme that realizes a given aspect matching approach. The same aspect could have multiple matching schemes. Hence, we need to capture more semantics about the matching schemes to know, which scheme can be applied in which context. However, this is not the end of the story, as each candidate needs to be evaluated against the required SLA. **3) Service Evaluation and Analysis:** Such stage takes the list of candidates from the previous state as input, then performs a more comprehensive Fit-GAP analysis on each candidate such as verifying the static QoS parameters against the required SLA, which is known as the service selection process. The output of this stage is an accurate list of atomic and composite services that can fulfill users' requirements. **4) Service Adaptation and Presentation:** Such stage takes the list of atomic and composite services from the previous stage as input, and checks for services that needs adaptation. It tries to automatically create the required service conversation adapters using different types of semantics as in [2]. The output of this stage is a list of customized services that exactly fulfill users' requirements. We include the adaptation stage as part of the discovery process, as there is no benefits from finding a service that cannot be used for heterogeneity reasons.

We argue that in order to build a customizable automated discovery process that could be dynamically constructed in a context-sensitive manner, users should be able to define any aspects they need in their queries. Also they should define any matching schemes and their arrangement to construct the required discovery process. To fulfill these requirements, we propose JAMEJAM, a framework for service discovery automation (i.e., named after the Persian myth of the divine cup that can provide answers for any question). It enables companies to create their discovery process as an executable BPEL process that describes the required matching aspects, matching schemes and matching policies. JAMEJAM realizes and executes such BPEL process by finding the suitable platform-services for every stage using different types of knowledge regarding: 1) the emergent and non-emergent aspects of the published services, 2) the aspects' description models, 3) the semantics of the services' application domains, 4) the semantics of the adopted matching approaches, 5) users' preferences, contexts and goals. JAMEJAM extends our previous work in [3] to include service evaluation and adaptation stages. We believe JAMEJAM is an essential platform-service needed for realizing the internet of services vision. JAMEJAM acts as a big tent for existing discovery approaches, as any existing service description model could be encapsulated in JAMEJAM via a software ontology, and any discovery approach could be encapsulated in JAMEJAM as a matching scheme. Experimental results show that JAMEJAM helps in increasing the accuracy and the adaptability of the discovery process. The rest of the paper is organized as follows. Section 2 provides an overview of the JAMEJAM framework. Section 3 explains how to use the framework. Section 4 presents the formal models needed by the framework; Section 5 discusses the framework query management. Section 6 discusses related work, while Section 7 summarizes the verification simulation experiments. Finally, Section 8 concludes the paper.

2. Jamejam Framework Overview

As we can see in Fig. 1, JAMEJAM mainly consists of four main subsystems: the aspects knowledge management subsystem, the services knowledge management subsystem, the matching schemes

knowledge management subsystem, and the service discovery subsystem. JAMEJAM subsystems also need a vertical layer of auxiliary services that help them to accomplish their tasks.

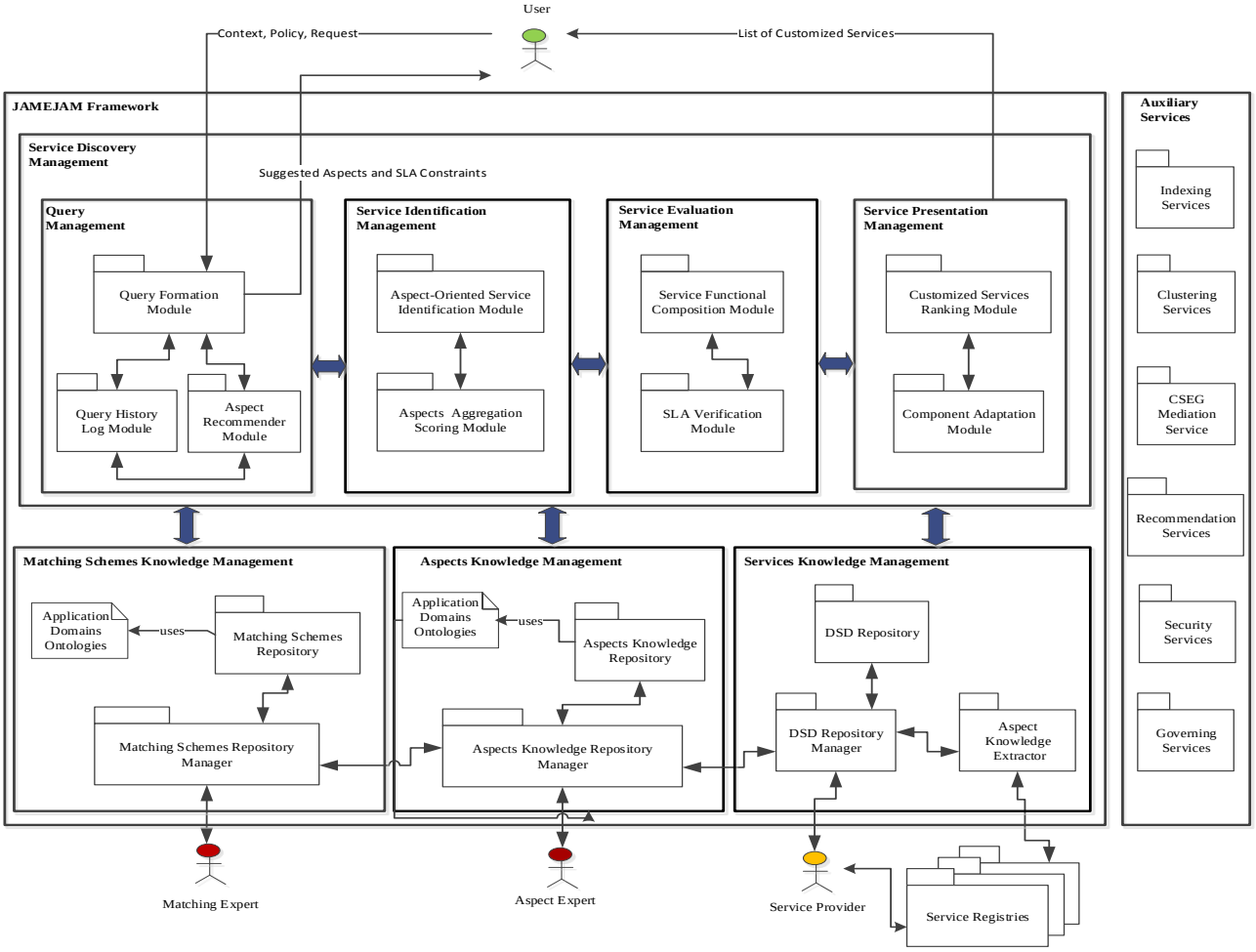

Fig. 1. JAMEJAM framework.

JAMEJAM subsystems could be summarized as follows: **1) Aspects Knowledge Management Subsystem:** It is the subsystem responsible for managing aspects knowledge, and its corresponding repository. An aspect knowledge is the facts, information, and skills acquired through experience, education, theory and practice regarding such aspect. JAMEJAM aims to capture such knowledge in a machine-understandable format following a dynamic, incremental, and context-sensitive manner, then stores such knowledge in a common repository, so that it can be shared between users and companies. One way to encapsulate such knowledge in a machine-understandable format is via ontologies. Hence, JAMEJAM enables the company experts to dynamically create their aspects' ontologies and aggregate them into software ontologies by providing the needed primitives, constructs, and infrastructure. Details are given in Section 4.

2) Service Knowledge Management Subsystem: It is the subsystem responsible for managing services knowledge, and its corresponding repository. Every service could have different semantic description models based on the software ontologies the service provider want to support, forming what is known by the service knowledge. JAMEJAM aims to capture such service knowledge in a machine-understandable format following a dynamic, incremental, and context-sensitive manner, then stores such knowledge in a common repository, so that it can be used during the service discovery process. Service creators should register their services with JAMEJAM by defining a dynamic service descriptor (DSD) for each service, which contains the aspects values defined according to the adopted aspects ontologies. JAMEJAM enables service

creators to enter the DSDs manually or automatically via extractors that retrieve the required knowledge from the service package. During the service discovery phase, only the aspect descriptors suitable for users' needs and goals will be used. **3) Matching Schemes Knowledge Management Subsystem:** It is the subsystem responsible for managing the matching schemes knowledge, and its corresponding repository. JAMEJAM aims to capture matching schemes knowledge in a machine-understandable format following a dynamic, incremental, and context-sensitive manner, then stores such knowledge in a common repository, so that it can be used during the service discovery process. **4) Service Discovery Management Subsystem:** It is the subsystem responsible for managing the service discovery process. Once a company defined its software ontology, matching schemes, and its services DSDs, it will be ready for service discovery. Such discovery management subsystem consists of other four subsystems: the query management subsystem, the service identification management subsystem, the service evaluation management subsystem, and service presentation management subsystem. **5) Auxiliary Services:** These are the services that JAMEJAM subsystems use to accomplish their tasks such as indexing and clustering services. We require the JAMEJAM framework to be deployed as a cloud platform service, so that it can be accessed by different user-bases.

3. JAMEJAM Usage Process

This section explains how users should use JAMEJAM. We summarize such usage process into the following main steps: **1) Models Preparation:** In this stage, the company experts should prepare models required for JAMEJAM processing. JAMEJAM requires different ontologies to work namely, application domains, matching schemes, and the matching aspects ontologies, which constitute the required software ontologies. Every company could create its own models and/or could select from existing ones. **2) Models Registration:** Once the company finalizes all the models required in the preparation stage, they should register these models with JAMEJAM (via a specific registration API). In this stage, the company experts should upload all the files corresponding to the defined ontologies and matching schemes**. 3) Models Values Extraction:** For every service registered with a given software ontology, JAMEJAM builds its dynamic service descriptor by invoking the defined extractors and/or interacting with the company users/experts to get the missing values. The service descriptors are created according to the different ontologies given in the registration stage. Once this step if finished, users will be able to use JAMEJAM for the registered software ontologies**. 4) Query Formation:** Users should submit their queries in a form of a JAMEJAM query, which is an XML file that contains their preferred matching aspects, preferred aspects' values, and preferred matching policy, then submit their query file to JAMEJAM (via a query API). Users can define any group of aspects they find suitable for their business need to be included in the query. However, they should specify only one matching policy per query, as this policy shows JAMEJAM the order in which the aspects will be processed and how their scores will be aggregated. If users are naive and cannot construct the JAMEJAM query, JAMEJAM has a recommendation system and templates that could be used to help them construct the JAMEJAM query, as shown in Section 5. If users requires the evaluation and adaptation stages to be incorporated, they have to explicitly mention that in their queries. Once JAMEJAM receives the query file, it constructs an executable BPEL process based on the defined aspects.

4. JAMEJAM Formal Models

In this section, we provide the required formal models needed to realize the framework.

A) Aspects Knowledge Formal Model: Company experts could define any aspects (emergent and non-emergent) that they find necessary to appear in their services' descriptions such as service goals, external behavior, supported interfaces, configurations parameters, usage statistics, maturity level, etc.

Once the company experts select the required aspects, JAMEJAM aims to capture such aspects knowledge in an ontology and stores such knowledge in a common repository, where aspect experts keep updating their knowledge in a cooperative and dynamic manner. JAMEJAM does not restrict aspect description to specific ontologies, but it allows a given aspect to be described using different ontologies capturing different semantics, provided that all of these ontologies are registered with JAMEJAM. Hence, JAMEJAM requires some other meta-data regarding the aspects given in the form of aspect descriptors. . Formally, JAMEJAM defines an aspect descriptor as a tuple <AspectName, Category, ADORef, AORef, SourcesRef, ExtractorsRef>, where AspectName is the name of the aspect according to the adopted application domain ontology referenced by ADORef. Generic names are assumed if there is no adopted application domain ontology, the value should set to Generic. Category specifies the service knowledge category the aspect belongs to, which should have one of the following four values: Nontechnical, High-Level-Functional, Low-Level-Functional, or Nonfunctional [1]. AORef is the reference for the ontology describing the aspect. SourcesRef specifies list of references to the required knowledge sources that indicate which files, documents, and information sources are needed to be packaged with the service by the service creators. . If there is no specific sources for such aspect, the aspect knowledge sources attribute should be set to the value Generic, and the aspect description value has to be manually entered by the service creators following the structure provided by the aspect ontology. ExtractorsRef is the list of references to the extractors that can be used to generate the aspect description from SourcesRef. This is important to automate the service knowledge extraction process, otherwise aspect descriptions will be entered manually by the service creators, and the value should be set to Manual. Of course, other meta-data attributes could be added to the JAMEJAM meta-model such as relations to other aspects, and adopted knowledge languages, however in this article, we focus only on those important attributes

Example 1: A Reputation Aspect. If a company likes to add a service reputation aspect to their services' descriptions such that it takes a value from 0 to 5, which represents the number of stars that consumers give to the service. The reputation aspect descriptor could be defined as <"Reputation", "Nonfunctional", "Generic", "http://www.semanticweb.org/ontologies/Reputation", "Generic", "Manual">, such aspect descriptor indicates that the reputation aspect is defined according to the referenced reputation ontology, and the aspect value entry is manual.

B) Matching Schemes Knowledge Formal Model: To register a matching scheme with JAMEJAM, matching experts are required to enter the corresponding matching scheme descriptor. We define a matching scheme descriptor as a tuple <SchemeRef, AspectName, ADORef, AORef, MatchingService, PreConditions, PostConditions, SchemeLogicType, SchemeApproach, ResultsExactness > such that SchemeRef is a unique reference for the matching scheme. AspectName is the name of the involved comparison aspect. ADORef is the reference to the adopted application domain ontology. AORef is the reference to the adopted aspect ontology. MatchingService is the reference to the web service encapsulating the matching approach. We require matching services to return a normalized matching score value (i.e. from the range [0..1]) for each examined service, where 0 means no match, 1 means perfect match, any value in between is a partial match (i.e., the greater the value, the closer the match). PreConditions is the set of preconditions that must be satisfied before invoking the matching service. PostConditions is the set of post-conditions that must be satisfied after successfully invoking the matching service. SchemeLogicType indicates if the matching scheme logic is syntactic or semantic (i.e. it takes on of the following values Semantic, Syntactic). SchemeApproach indicates if the matching scheme is structured or generic (i.e. it takes on of the following values Structured, or Unstructured). ResultsExactness indicates if the obtained matching results are exact or approximate.

Example 2: A Behavior Matching Scheme. For example, the company experts need to match the

behavior aspect using the semantic approach discussed in [1], hence, the semantic behavior matching scheme should be defined as <BehaviorSemanticScheme, ExternalBehavior, "http://www.semanticweb.org/Elgedawy/Ontologies/Banking-ver2-2008", "http://www.semanticweb.org/Elgedawy/Ontologies/GPLUS-ver1","http://www.WebServices.org/Elgedawy/Schemes/SMP-Matching",{},{}, Semantic, Structured, Exact >. This enables the company to create its matching knowledge repository. Once the matching repository is built, matching experts could compare approaches' performance and accuracy.

C) Software Ontology Formal Model: Once the company experts defined their preferred aspects and matching schemes, they can group these definitions into software ontologies. A software ontology is simply a collection of aspects and matching schemes descriptors defined before. We formally define the software ontology as the tuple <SWORef, AspectsList, MatchingSchemesList>, where SWORef is a reference to the software ontology, AspectsList is a list of required aspect descriptors, and MatchingSchemesList is a list of the required matching schemes descriptors. Use of software ontologies will make things easier for the users, as they just need to reference the required software ontology in their queries, and JAMEJAM will simply know the involved aspects and matching scheme definitions.

D) Service Knowledge Formal Model: Service knowledge is the collective knowledge regarding the service various aspects. That for every aspect defined in the aspect knowledge module, a corresponding value should appear in the service description. Such service description should be captured in a machine-understandable format so that it can be understood by the discovery agent. Such service description is dynamic as aspects' values may change overtime, also aspects could be added/removed from the JAMEJAM framework according to the adopted company policies. Hence, every service registered with JAMEJAM should have a dynamic descriptor known as a DSD (i.e., Dynamic Service Descriptor). We define the service DSD as set of aspect value descriptor, where an aspect value descriptor is defined as the tuple < AspectName, Category, ADORef, AORef , AspectValue >, where AspectName is the name of the aspect, Category specifies the service knowledge category that the aspect belongs to, ADORef is the reference to the adopted application domain ontology, AORef is the reference for the ontology describing the aspect, and AspectValue is the aspect value according to the adopted aspect ontology. Such DSDs should be defined in any machine-understandable format such as XML format. Every aspect value is defined according to its corresponding aspect ontology. For example, business scopes are defined based on the ontology given in [4]. Once the service DSD is created, it is stored in the DSD repository so it can be used during the service discovery process. JAMEJAM stores services DSDs in a different repository from the services repositories, so it can have total control over the services DSDs, as service repositories could be external to JAMEJAM.

5. Query Management

Users should define the required query aspects and their values, their contexts, goals, and SLA obligations. Also they should define the required matching policy. The notion of a matching policy is introduced to enable users to define their preferences and logic regarding the service identification process. The matching policy contains the logical constructs of how to use the aspects' matching schemes to build the identification process. Users can arrange such matching schemes in any way they find suitable. Such matching policy could be easily described as an abstract BPEL process, as the BPEL language is powerful enough to express complex processes, where the BPEL partners will be the retrieved platform-services corresponding to the identified matching schemes. Once all this information are defined in the JAMEJAM query, users should submit their query in a machine-understandable format. For simplicity, we propose to use standard XML format for defining the aspects and their required attributes' values, as shown in Fig 2. The figure shows different aspects to be used in the search process, also it shows the required matching

policy and the matching schemes' preferences. The query contains a goal and targets business scope, external behavior, and reputation aspects. Every aspect is defined according to its corresponding ontology. The required aspects, their corresponding descriptors, the adopted application domain and aspect ontologies are encapsulated in the adopted software ontology. Users could directly define the required software ontology, or just refer to the required aspects and application domain ontologies. If no ontologies are defined, the query is assumed generic, and generic matching schemes are used to match the aspects. For naïve users, the query management system could provide recommendations and query templates for the users to create the query.

```
<Query>
      <AspectName=``BusinessScope", Category= ``Nontechnical", ADORef=`../>
      <AspectName=``Reputation", Category= ``Nonfunctional", ADORef=``>      <
        Condition>      <Comparator>    GTE    </Comparator>      <Value>  2    </Value>
        </Condition>
      </Aspect>
      <AspectName=``Behavior", Category= ``High-Level-Functional",                ADORef=``...",
      AORef =``..." >
                        <Operation    >
                              <Inputs> ……</Inputs>
                              <Outputs> ……</Outputs>
                               <PreConditions> ….</PreConditions>
                               <PostConditions> ….</PostConditions>
                           </Operation>
       </Aspect>
       <Matching Policy, method = ``Hierarchal ">
              <Sequence>
                        <MatchingScheme      stage=Identification >
                                  <AspectName>                ``BusinessScope"
</AspectName>
                                  <SchemeLogicType>                Syntactic
</SchemeLogicType>
                                  <SchemeApproach>                Structured
</SchemeApproach>
                                  <ResultsExactness>                Exact
</ResultsExactness>
                        </MatchingScheme>
                         <MatchingScheme      stage=Identification >
                                  <AspectName> ``Reputation" </AspectName>
                                  <SchemeLogicType>                Syntactic
</SchemeLogicType>
                                  <SchemeApproach>                Structured
</SchemeApproach>
                                  <ResultsExactness>                Approximate
</ResultsExactness>
                        </MatchingScheme>
                         < MatchingScheme       stage=Evaluation>
                                  <AspectName> ``Behavior" </AspectName>
                                  <SchemeLogicType>                Semantic
</SchemeLogicType>
                                  <SchemeApproach>                Structured
</SchemeApproach>
                                  <ResultsExactness>                Exact
</ResultsExactness>
                        </MatchingScheme>
              </Sequence>
       </Matching Policy>
      <Correctness Criteria>      ……      </Correctness Criteria>
      ...
 </Query>
```

Fig. 2. JAMEJAM query example.

6. Related Work

Currently, we did not find any existing automation framework that covers all the stages of the service discovery process. However, existing approaches covered some stages of the discovery process. Hence, in Table 1, we provide a comparison of some of the existing approaches with JAMEJAM in terms of their stages coverage. We consider a stage is fully covered if the approach proposed a semantic solution for the stage problem. However, the stage is partially covered if the proposed solution is approximate (i.e., only addressing one aspect) or generic (i.e., adopts only keywords). As we can see in the table, only JAMEJAM managed to cover all the discovery process stages.

Table 1. Service Discovery Process Realization Comparison

Works	Service Description	Service Identification	Service Evaluation	Service Selection	Service Adaptation
[1]	Full	Full	Full	Partial	
[3]	Full	Full			
[5]	Partial	Partial	Partial		
[6]	Full	Partial	Full	Partial	
[7]	Full	Full	Full	Partial	Partial
[8]	Partial	Partial	Partial		
[9]	Full	Partial	Full	Partial	
[10]	Full	Partial	Full	Partial	
JAME JAM	Full	Full	Full	Full	Full

7. Experiments

This section provides information regarding the simulation experiments performed to verify JAMEJAM's adaptability and accuracy. Hence, we create different usage scenarios and compare the resulting discovery accuracy using the well-known precision and recall metrics, and compute the time taken to construct the discovery process to check adaptability speed. However, lack of real life data that contains semantic descriptions for services still a big challenge for researchers till today [1], [5], [6]. Hence, researchers opt to use artificial data for their experiments. Such approach has been widely adopted by many works such as the works in [1], [5], [6]. Hence, in this article we will follow the same approach and generate the artificial data suitable for our experiments.

We adopt the same steps used to generate artificial data mentioned in [1], [3]. However, due to space limitation we will no mention the steps here, and interested readers will find the details in [1], [3]. In this dataset, we generate DSD for a number of services (i.e. arbitrary chosen as 10,000 service), such DSDs contain the business scope aspect (described as in [4]), the behavior aspect, and the reputation aspect. The behavior models are extracted from defined operations' sequences and concepts obtained from a generated artificial application domain ontology [1]. The corresponding matching approaches described in [1] and [4] are encapsulated as services. To generate the query set. We select a random 100 distinct service DSD from the generated dataset. For each DSD in the query set, we generate a random number of DSD replicas. Such number is chosen from the arbitrary range of (0-50) to ensure having different number of services for each service description. Finally, such generated replicas are added to the dataset and randomly distributed among the DSDs. By doing so, we can automatically identify the correct answer for each query, which is the corresponding service and its replicas. Hence, recall and precision could be automatically computed. Once the dataset and query sets are generated, we insert the required matching policies in the queries, so we can see how JAMEJAM will react. The first matching policy scenario, we required the aspects to be matched in a cascade order starting by the business scope aspect, followed by behavior aspect, followed by the reputation aspects, then we computed the corresponding precision and recall, as shown in Fig. 3. That

shows by combining the three approaches using JAMEJAM, we managed to increase the discovery accuracy compared to the cases when only one aspect is used, as combining between different aspects minimizes the chance for the appearance of false positives. The discovery process construction took less than 1 sec, as the matching schemes repository were indexed and quite small in size, as most of the time is consumed searching such repository.

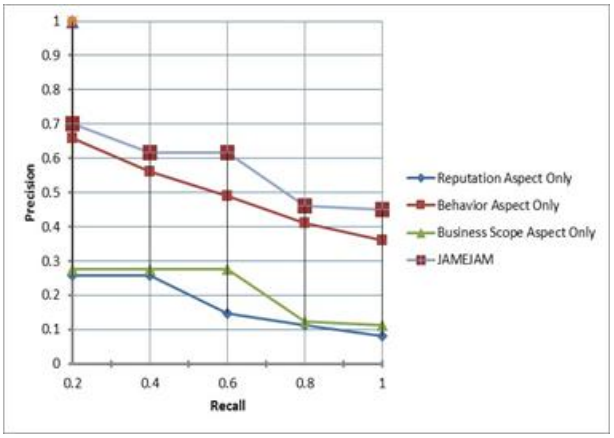

Fig. 3. Cascading matching policy.

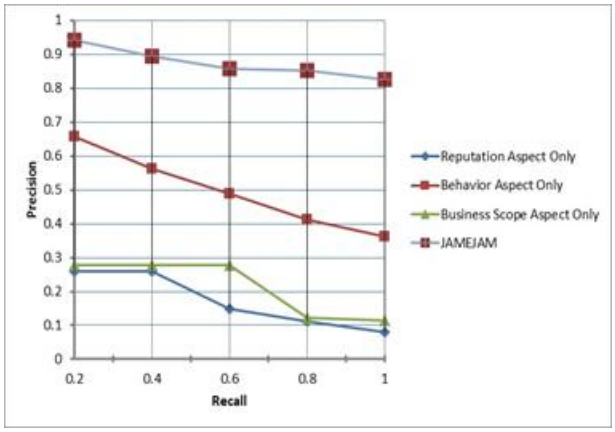

Fig. 4. Weighted matching policy.

To show the adaptability of JAMEJAM, we repeated the same experiments but with using a weighted matching policy scenario, in which all the matching schemes are invoked in parallel, and the final obtained matching results scores are aggregated based on matching schemes weights (i.e., behavior weight = 0.5, scope weight= 0.3, and reputation weight =0.2), then we computed the corresponding precision and recall, as shown in Fig. 4 just by changing the query, the whole discovery process will change accordingly. Discovery process construction time did not change from the previous case, however Fig. 5 shows that by combining the three approaches using JAMEJAM, we also managed to increase the discovery accuracy compared to individual aspects cases. However, we can notice, the weighted matching policy performed better than the cascading matching policy for the generated data set. Such information is obtained due to the help of JAMEJAM, which provides the infrastructure required for service discovery analytics. Hence, we can say JAMEJAM can help users to customize their discovery process to obtain the best discovery accuracy by trying different discovery process configurations (i.e., aspects, matching schemes, and matching policies) over their data, then choose the best performing configuration.

8. Conclusion

In this paper, we proposed JAMEJAM, an automation framework for the service discovery process that enables users to create customizable discovery processes on the fly. This is done by adopting different types of knowledge namely: application domains knowledge, aspects knowledge, services knowledge, matching schemes knowledge, and discovery process knowledge. We discussed the framework's main components and provided the required formal models. Experimental results show that JAMEJAM helps in increasing the accuracy and the adaptability of the discovery process.

References

[1] Elgedawy, I., Tari, Z., & Thom, J. A. (2008). Correctness-aware high-level functional matching approaches for semantic web services. *ACM Transactions on Web, Special Issue on SOC, 2(2)*.

[2] Elgedawy, I. (2011). On-demand conversation customization for services in large smart environments. *IBM Journal of Research and Development, Special issue on Smart Cities, 55(1/2)*.

[3] Elgedawy, I. (2015). USTA: An aspect-oriented knowledge management framework for reusable assets discovery. *The Arabian Journal for Science and Engineering*, *40(2),* 451-474.
[4] Elgedawy, I., & Ramaswamy, L. (2009). Rapid identification approach for reusable soa assets using component business maps. *Proceedings of IEEE 7th International Conference on Web Services.*
[5] Bislimovska, B., Bozzon, A., Brambilia, M., & Fratetnali, P. (2014). Textual and content-based search in repositories of web application models. *ACM Trans. Web, 8(2)*, 11:1–11:47.
[6] Kririkos, K., Plexousakis, D., & Paterno, F. (2014). Task model-driven realization of interactive application functionality through services. *ACM Trans. Interact. Intell. Syst, 3(4)*, 25:1–25:31.
[7] Roman, D., *et al.* (2005). Web service modeling ontology. *Applied Ontology, 1(1),* 77–106.
[8] Bianchini, D., *et al*. (2014). Service identification in interorganizational process design. *IEEE Transactions on Services Computing, 7(2)*, 265–278.
[9] Zisman, A., Spanoudakis G., Dooley, J., & Siveroni, I. (2013). Proactive and reactive runtime service discovery: A framework and its evaluation. *IEEE Transactions on Software Engineering, 39(7),* 954–974.
[10] Brogi, A., Corfini, S., & Popescu, R. (2008). Semantics-based composition-oriented discovery of web services. *ACM Trans. Internet Technol, 8(4),* 19:1–19:39.

Islam Elgedawy is an associate professor at the Computer Engineering Department, Middle East Technical University, Northern Cyprus Campus. He received his B.Sc. and M.Sc. degrees in computer science from Alexandria University, Egypt in 1996, and 2000, respectively, and his Ph.D. degree in computer science from RMIT University, Australia in 2007. His work focuses on the areas of service-oriented computing, organic computing, and software engineering. He has a growing record of international publications, consultancy and professional services.