NLSC: Unrestricted Natural Language-based Service Composition through Sentence Embeddings

Oscar Javier Romero López, Ankit Dangi and Sushma Anand Akoju

January 30, 2019

NLSC: Unrestricted Natural Language-based Service Composition through Sentence Embeddings

1st Author
Department
Organization
email address

2nd Author
Department
Organization
email address

3rd Author
Department
Organization
email address

***Abstract*—Current approaches for service composition (assemblies of atomic services) require developers to use: (a) domain-specific semantics to formalize services that restrict the vocabulary for their descriptions, and (b) translation mechanisms for service retrieval to convert unstructured user requests to strongly-typed semantic representations. In our work, we argue that effort to developing service descriptions, request translations, and matching mechanisms could be reduced using unrestricted natural language; allowing both: (1) end-users to intuitively express their needs using natural language, and (2) service developers to develop services without relying on syntactic/semantic description languages. Although there are some natural language-based service composition approaches, they restrict service retrieval to syntactic/semantic matching. With recent developments in Machine learning and Natural Language Processing, we motivate the use of Sentence Embeddings by leveraging richer semantic representations of sentences for service description, matching and retrieval. Experimental results show that service composition development effort may be reduced by more than 44% while keeping a high precision/recall when matching high-level user requests with low-level service method invocations.**

***Index Terms*—Service composition, Middleware, Sentence Embeddings, Named-Entity Recognition, Effort Estimation**

I. Introduction and Related Work

Service is any software component, data, or hardware resource on a device that is accessible by others [6]. *Service composition* is the process of aggregating such reusable atomic services to create complex compositions. Existing research can be seen in two directions, where, (a) both atomic and composite services are defined using description languages (such as BPEL4WS, OWL-S, and WSDM) in terms of service input/output, pre- and post-conditions, fault handling and invocation mechanisms. Such service descriptions serve as inputs to orchestration engines [30], [32] that generate declarative specification of workflows to compose different services; and (b) architectural middlewares [6], [14], [19], [25] that assume a declarative specification of a composition. A substantial amount of effort is required in both directions to define and integrate services, mainly due to: (a) the use of domain-specific languages and semantics for service descriptions and compositions; (b) strongly-typed orchestration languages (e.g., BPEL, WSDL, OWL-S, etc.) restricting heterogeneous service composition; (c) statically specified compositions that create design-time couplings preventing dynamic adaptation; and (d) there is more than one composite service description languages: different ontologies have been designed resulting in different vocabularies, thwarting true semantic interoperability, so technologies have yet to converge and standardize [31]. In natural language-based service composition middleware, end-users interact instinctively with systems in natural language and expect the system to identify services that meet their goals. These kind of middleware can be broadly categorized as those that: (a) apply restrictions on how the user expresses the goal with sentence templates and then use structured parsing to match against service descriptions [4], [23]; (b) construct semantic graphs to represent service descriptions and match against a lexical database such as WordNet to compute concept similarity [9], [13], [26]; and (c) match partially-observable natural language request with semantics of service description expressed using semantic web services (OWL-S, VDL) [8], [27]. Limitations with these approaches include: (a) complex linguistic processing that requires additional natural language processing (NLP) techniques: structured parsing, extracting parts-of-speech, stop-word removal, spell-checking, stemming, and text segmentation; (b) inclusion of lexical databases such as WordNet or domain-specific ontologies; and (c) a weaker concept representation and similarity score for semantic matching that does not account for sentence context.

Research Questions: to overcome the above, we address:
RQ1: How to reduce the amount of development effort and complexity to develop service compositions?
RQ2: How can both end-users and developers create service compositions in an intuitive, efficient, and dynamic way using natural language-based descriptions?

Main Contributions: we address RQ1 by removing effort-consuming engineering practices, such as: (a) formal service descriptions that use syntactic/semantic representations; and (b) orchestration processes that use domain-specific languages. Additionally, we provide an automated OSGi-based toolchain for service modularity, service discovery, service deployment, and service execution. Our toolchain allows transparently deploying OSGi components to either cloud-based applications or mobile Android-based apps. And we addresses RQ2 by developing *NLSC*, a Natural Language-based Service Composition Middleware that: (a) allows users to express template-free service requests using natural language without complex linguistic processing; (b) avoids the need for lexical databases, semantic graphs, and domain-specific ontologies; (c) generates dynamic service compositions by directly binding high-level user requests to low-level service invocations without having to define ontological service descriptions or strongly-typed well-defined interfaces; and (d) uses a stronger representation of sentence semantics to characterize words and concepts that account for word usage in context to the sentence by applying a state-of-the-art pre-trained semantic representation model of English language. The remaining of this paper is organized as follows: Section II presents the background and motivating example. Section III details the design and implementation and Section IV reports the experimental results. We introduce the related work and conclude the paper in Section V and Section VI, respectively.

II. Background and Motivation

A. Service Composition Middleware Model

According to the Service Composition Middleware (SCM) model [16] (a high-level abstraction model that does not consider a particular service technology, language, platform or algorithm used in the composition process), middleware for service composition can be largely classified into four main modules as follows: *Translation, Generation, Evaluation*, and *Execution*. In SCM, applications may send requests to middleware using diverse specification languages or techniques, and the *Translator* converts these request descriptions into a system comprehensible language (i.e., formal languages and models) that can be used by the middleware. Once translated, the request specification is sent to the Generator, which provides the needed functionality by composing the available services, and generating one or several composition plans. This service composition is technically performed by chaining interfaces using either a syntactic or semantic method matching (or both). Then, the Evaluator chooses the most suitable composition plan depending strongly on many criteria like application context, the service technology model, the non functional service QoS (Quality of Service) properties, etc. Finally, the Builder executes the selected composition plan and produces an implementation corresponding to the required composite service. Once the composite service is available, it can be executed by the application that required its functionality.

B. Motivational Example

Suppose the user is planning a trip to Paris on a specific range of dates (main goal) using a smartphone that does not have a *trip planner* service or app installed. This main goal can be decomposed into sub-goals such as: (1) check schedule availability on dates, (2) look for flights cheaper than \$700, (3) book the chosen flight, (4) search for hotels under \$100/night near downtown, (5) book the selected hotel, (6) check the weather conditions for given dates, (7) if weather conditions are bad, look for indoor activities to do, (8) otherwise, look for outdoor activities to do. To address this scenario (see Figure 1), a *service developer* would create atomic service interfaces and implementations for services such as Maps, Calendar, FlightBooking etc.; a *service modeler* would define the service interface contracts using WSDL; an *ontology engineer* would maintain the trip-planning ontologies and ensure consistency

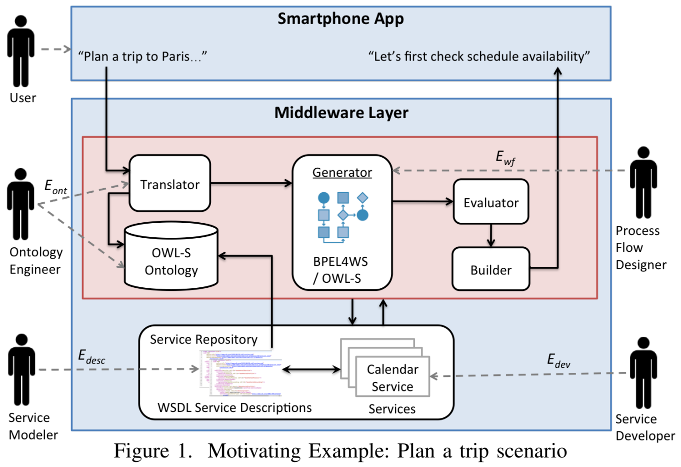


Figure 1. Motivating Example: Plan a trip scenario

with OWL-S models; and a *process flow designer* would investigate explicit declarative alternatives to generate a service composition that addresses user's goal.

C. Problem Statement

In the example above, the total effort required is the sum of partial efforts. Let $E_T = \sum_{i=1}^{n} E_i$, where, $E_i \in \{E_{dev}, E_{desc}, E_{ont}, E_{wf}\}$ such that E_{dev}, E_{desc}, E_{ont}, E_{wf} are effort amounts to develop a service implementation, generate a WSDL service description, create/maintain an OWL-S ontology, and maintain a BPEL4WS workflow, respectively (for simplicity, we ignore additional efforts for testing, CI/CD, etc.) These efforts increase exponentially when service requirements change continuously, there are inconsistencies on service contracts, or developers don't have the proper skillset.

D. Goals

Our scientific intuition leads us to hypothesize that a data-driven approach (using large text corpus and datasets of common-sense sentences) not only could minimize the effort of developing service compositions by removing the need of specifying strongly-typed, syntactically/semantically well-defined, domain-dependent service descriptions, but also could outperform traditional semantic-driven approaches that require continuous validation of consistency due to human designers' biased models. Therefore, driven by our RQs, our goal is two-fold: (a) to reduce the total effort of integrating new services into a composition by merging/replacing some of the development tasks previously described without affecting either system performance or the quality of service compositions; and (b) to automatically bind unrestricted natural language user requests to unstructured natural language service descriptions with control structures for composition.

E. OSGi

OSGi (Open Services Gateway initiative) technology [1] is a set of specifications that define a dynamic component system for Java. These specifications enable a development model where an application is composed of several components that are packaged as bundles. Components communicate locally and across the network through services. Services have an API that is defined in a Java package. Some of the most known

OSGi-based middleware for service composition are: [15], [20], [28], [29]. We use OSGi as a backbone for connecting multiple service implementations, providing a mean for the exchange of information between them.

III. Approach

A. Preliminaries

As it is generally considered in the literature [3], [32], we distinguish two types of services [3], [32]: *abstract* and *concrete* services. Formally, a concrete service cs_i is a tuple $\langle cs_i^{in}, cs_i^{out}, cs_i^{prec}, cs_i^{postc}, cs_i^{QoS}\rangle$ that performs a functionality by acting on input data (cs_i^{in}) to produce output data (cs_i^{out}), with pre-conditions (cs_i^{prec}), post-conditions (cs_i^{postc}) and Quality of Service (cs_i^{QoS}) requirements. An abstract service as_i is a tuple $\langle as_i^{in}, as_i^{out}, as_i^{cs}\rangle$ realized by several concrete services $as_i^{cs} \in \{cs_{(i,1)}, cs_{(i,2)}, ..., cs_{(i,n)}\}$ that offer the same functionality with input parameters (as_i^{in}), output parameters (as_i^{out}) such that $\forall cs_{(i,j)}, cs_{(i,k)} \in as_i^{cs}/(as_i^{in} = cs_{(i,j)}^{in} \cap cs_{(i,k)}^{in}) \wedge (as_i^{out} = cs_{(i,j)}^{out} \cap cs_{(i,k)}^{out})$.

B. Reference Architecture

Figure 2 presents a reference architecture for service composition that will help highlight where our contributions lie. *Step 1:* service developers continuously implement, integrate, deploy and publish service components (either abstract or concrete). Developers add unstructured and unrestricted natural language descriptions (in the form of plain code annotations) to each single service component and its atomic methods. In comparison, traditional approaches would include additional steps (i.e., service description using WSDL, creation and validation of OWL-S ontology, etc.). *Step 2:* an automated process extracts those descriptions from the code annotations and puts them on a separate repository. *Step 3:* the end-user, an application developer, or a top-tier application makes a service request (e.g., *"I want to plan a trip to Paris from Sept. 29 to Oct. 11"*). *Step 4:* a coordination system is in charge of orchestrating the high-level assembly of abstract services by chaining service pre- and post-conditions and matching data types (traditional approaches would include additional steps for creating complex graph-based, workflow-based or rule-based plans). *Step 5:* the service matching is performed using two NLP techniques: Sentence embeddings and Named-Entity Recognition, and returns a set of abstract services and their corresponding concrete service candidates. Steps 4 and 5 are repeated until a composition plan is tailored. *Step 6:* a mechanism validates the QoS requirements by selecting a sub-set of candidate concrete services that are to be executed. *Step 7:* using a service discovery mechanism, the sub-set of concrete service candidates are looked up in the registry and service availability for those is confirmed. *Step 8:* lastly, a composite service is generated and executed. Compared to the SCM model, we suppress the Translator module and only keep the remaining ones (Generator, Evaluator, and Builder). We overcome the need for the Translator as our approach does not use intermediate representations such as ontologies, graph-based models, and so forth, rather, we provide a direct

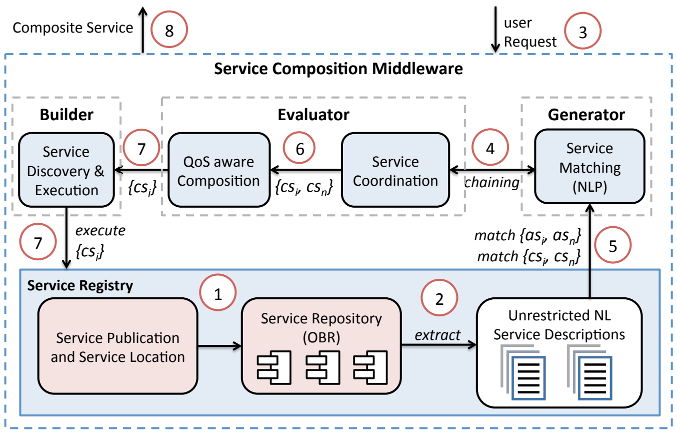


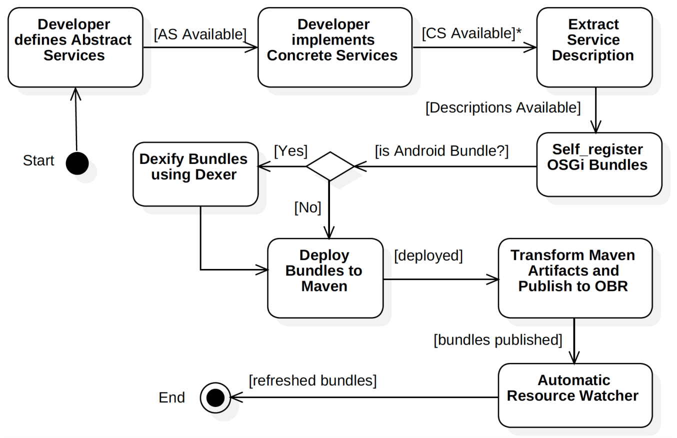
Figure 2. Reference Architecture for a NLSCM

correlation between the request and the service description which are both expressed using unrestricted natural language.

C. Service Development Process

NLSC was architected to be deployed on distributed environments and to support different kind of client applications (e.g., standalone, web, mobile, etc.). This requirement, along with dynamicity, low latency, high-performance, modularity, and support to Android Runtime Environment (ART), were the main architectural significant requirements we took into consideration over its implementation. Extending our initial definition of "service", we consider that an Android-based service can be a device sensor, a local service (installed on the Android device), or an app, hence, we avoid making the assumption that only a web service model should be applicable to *NLSC*. Given these requirements, a suitable solution for developing dynamic service components for Android is the OSGi technology [18]. Most of the OSGi-Android approaches [5], [7], [11] are based on Apache Felix [2], an implementation of the OSGi Framework and Service platform. In *NLSC*, composite services are created from a dynamic assembly of black box components, executing in a local Felix container, which does not provide mandatory non-functional services. Services do not contain any reference between them at design-time, and respect black box and late-binding concepts. We tried to keep the intervention of application developers to the minimum, automating as much as possible the discovery, composition, invocation and interoperability of services and, therefore, reducing the development effort. To that purpose, we developed a set of tools to simplify the process of service development, and promote agile development and continuous integration into *NLSC*. This process is depicted in Figure 3 and described below.

1) Abstract Services Description: Common OSGi-based approaches for service composition provide modularity, though still a Translator is required to guarantee interoperability between semantic and syntactic service description languages that are both heterogeneous. Prior work shows the heavy cost of the syntactic and semantic matching [16]. In our work, we replace effort-consuming syntactic/semantic service descriptions (WSDL, OWL-S, etc.) by intuitive code



Developer defines Abstract Services
[AS Available]
Developer implements Concrete Services
[CS Available]*
Extract Service Description
[Descriptions Available]
Self_register OSGi Bundles
[is Android Bundle?]
[Yes]
Dexify Bundles using Dexer
[No]
Start
Deploy Bundles to Maven
[deployed]
Transform Maven Artifacts and Publish to OBR
[bundles published]
Automatic Resource Watcher
[refreshed bundles]
End

Figure 3. Workflow for Service Development using *NLSC*

annotations that allow developers to add unrestricted natural language descriptions to each service component and its methods. More specifically, we provide 2 code annotations: *@Description* annotation allows developers to add a set of possible capabilities for an atomic service method (in terms of what the service method can do), and *@ArgDesc* let developers add descriptions to method arguments that can be later used for argument type disambiguation. Alternatively, *NLSC* can also load these annotations from plain files in order to decouple them from the programming language.

```
public interface CalendarService {
  @Description(capabilities = {
      "validates availability on calendar given a time slot",
      "checks conflicts on calendar given a range of dates",
      "checks calendar availability on a range of dates"
  })
  @ArgDesc(arguments = {
      "fromDate : check calendar from date (yyyy-mm-dd)",
      "toDate   : check calendar to date (yyyy-mm-dd)"
  })
  Boolean checkAvailability(Date fromDate, Date toDate);
}
```

Listing 1: Abstract service description for CalendarService

From Listing 1, it is worth noting that: a) given an abstract service as_i exposes a set of methods $m_i = \{m_{(i,1)}...m_{(i,n)}\}$, where, a method is described as a tuple $\langle cap_{m_i}, args_{m_i}, argd_{m_i}\rangle$ where cap_{m_i} is an arbitrary number of capabilities such that $|cap_{m_i}| \geq 1$, $args_{m_i}$ is a set of method arguments, and $argd_{m_i}$ is a set of argument descriptions corresponding to $args_{m_i}$, such that $|args_{m_i}| = |argd_{m_i}| \geq 0$; b) the list of arguments in @ArgDesc maps each description to a method argument using the name of the argument, a description in natural language, and (optional) the format or type of the argument (used to disambiguate with the user or when using the Named-entity Recognition technique). c) developer does not need to do extra effort defining a service description or ontology using WSDL, OWL-S, etc. (especially when developers are unfamiliar with such languages, but even if they are, using unrestricted natural language descriptions is more intuitive and easy-to-deploy).

2) Concrete Services Implementation: A concrete service cs_i inherits method descriptions from abstract service as_i. It defines non-functional, platform-specific QoS requirements for methods to guarantee service execution if and only if they are met. For illustration purposes, let's continue with our motivating example and assume the platform is Android. An abstract service could be *CalendarService* (as_{cal}) whereas concrete services could be *GoogleCalendarService* (cs_{gc}) and *YahooCalendarService* (cs_{yc}). *NLSC* provides a set of pre-defined QoS annotations for Android, though they can extended: @BatteryQoS is a categorical value for battery level consumption that indicates whether the service is battery-intensive (e.g., LOW_BATTERY, HALF_CHARGED, FULLY_CHARGED), @ConnectivityQoS is a categorical value that determines whether the service requires deveice's wifi connection or if it runs locally, etc. From Listing 2, we observe that YahooCalendarService will be executed only if its QoS features are met, that is, the smartphone's battery has to be at least half charged and it should be connected to the WiFi, otherwise, another concrete service that implements CalendarService is discovered.

```
public class YahooCalendarService implements CalendarService{
  @BatteryQoS( minBattery = Constants.REQUIRES_HALF_CHARGED )
  @ConnectivityQoS( wifiStatus = Constants.REQUIRES_WIFI )
  public Boolean checkAvailability(Date from, Date to) {
        Log4J.info("Executing YahooCalendarService....");
  }
```

Listing 2: QoS-awareness for YahooCalendarService

3) Service Descriptions Extractor: Using Java reflection, this tool automatically generates a plain file with all service method descriptions which is further used by the Service Matching module (Section III-D). Additionally, it generates a metadata file with method argument descriptions and QoS values that are used at the time of service execution after services are grounded (Section III-F).

4) OSGi bundle self-registration: Both abstract and concrete services are deployed as OSGi bundles. This tool automatically generates an implementation of a BundleActivator (an OSGi interface that manages bundle's lifecycle) and injects code on the *start()* and *stop()* methods to self-register or self-unregister the bundle against the Felix Framework.

5) Dexifying bundles: Android Runtime does not use Java bytecode, instead, Android programs are compiled into .dex (Dalvik Executable) files. Thus, we developed Dexer, a tool that automatically transforms the Java class files compiled by a regular Java compiler into a class file format that can be executed on the Android runtime. In other words, Dexer automatically converts an OSGi bundle into an executable Jar that can be later executed on the Android platform.

6) OSGi Maven Deployer: Transforms application Jars to OSGi bundles that are then automatically deployed to a remote Maven repository, which makes the artifacts accessible to application developers and service runtime environment.

7) TAMO: This is a tool that automatically transforms artifacts from a Maven repository (that holds OSGi bundle artifacts) to an OSGi Bundle Repository (OBR). Felix OBR provides a service that can automatically install a bundle, with its deployment dependencies, from a bundle repository, enabling location and discovery of the participating services during the composition process.

8) ARW: The Automatic Resource Watcher (ARW) pulls data periodically from an OBR in order to find new available services or updates for existing services. This functionality is critical for the service discovery phase during service execution because it allows re-configuration of services and enables the generation of compositions on-demand.

D. Service Matching

Current approaches on service composition perform service matching by doing syntactic and semantic interface matching, then the service evaluation is performed upon the input/output matching correctness. As we described before, semantic matching though useful is expensive in terms of effort (ontology designers have to validate the consistency of semantic representations) and computing time (the larger an ontology is, the longer it takes to perform semantic inference or concept graph search). Instead of using syntactic or semantic matching through the use of ontologies, we propose semantic service matching through the use of *Sentence Embeddings*. In linguistics, and more specifically in feature learning techniques in natural language processing (NLP), both word embeddings and sentence embeddings are discussed in the research area of distributional semantics. Embeddings aim to quantify and categorize semantic similarities between linguistic items based on their distributional properties in large samples of language data. Word embeddings capture the idea that is possible to express "meaning" of words using a vector, so that the cosine of the angle between the vectors captures semantic similarity. ("Cosine similarity" property.) Sentence embeddings and text embeddings extend word embeddings to sentence and larger pieces of text: use a fixed-dimensional vector to represent a short piece of text, e.g., a sentence or a small paragraph. In order to perform text understanding using sentence embeddings we use *sent2vec* [24], a model that can be seen as an extension of the CBOW (Continuous Bag of Words [21]) where the training objective is to train sentences instead of word embeddings. Sentence embeddings account for sentence context with the words in the sentence as compared to previous natural-language based service matching where only word-level matching is performed ignoring the context. Such an embedding provides a richer semantic representation that makes it a natural choice for using natural-language descriptions for service matching.

Let's return to our motivating example. Assume that one of the steps for achieving user's goal (*"plan a trip to Paris..."*) is the user request *"check what's on my schedule from Sept. 29 to Oct. 11?"* as shown in Figure 4. In order to find the nearest neighboring sentence feature (the optimal match in terms of a higher sentence embedding match), it is necessary to provide both a pre-trained model learned using unsupervised learning over a large dataset of sentences (19.7 billion sentences) and a corpora in which it is possible to search for the

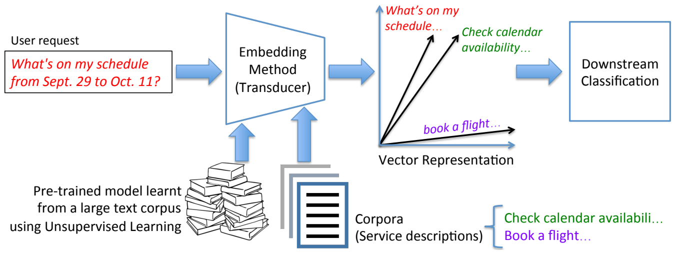

User request
What's on my schedule from Sept. 29 to Oct. 11?
Embedding Method (Transducer)
What's on my schedule...
Check calendar availability...
book a flight...
Vector Representation
Downstream Classification
Pre-trained model learnt from a large text corpus using Unsupervised Learning
Corpora (Service descriptions)
Check calendar availabili...
Book a flight...

Figure 4. Pipeline for the Service Matching using Text Embedding

nearest neighboring sentence to the given input sentence (user request). This corpora corresponds to the service descriptions that have been previously extracted (see Section III-C3). Input sentences (both user requests and services descriptions) are represented in the vector space and the semantic similarity is computed after executing the downstream classification, that is, similar sentences such as user request *"what's on my schedule..."* and a service description *"check calendar availability on dates..."* for method *checkAvailability()* (in abstract service CalendarService) will be closer whereas the similarity between the same user request and method description *"book a flight..."* from FlightReservationService will be farther (the mathematical model can be found in [24]).

E. Service Coordination

The Service coordination comprises three mechanisms: a rule-based system that allows creating high-level assemblies of abstract services by chaining pre/post-conditions, a short-term (working) memory where results are stored temporarily, and a Named-entity extractor for data types and entity matching. As a rule-based system we use *easy-rules* [10], a lightweight yet powerful Java rule engine that can be executed in a wide variety of platforms, including Android. easy-rules also supports MVEL (MVFLEX Expression Language [22]), a hybrid dynamic/statically typed, embeddable Expression Language and runtime for the Java Platform. MVEL is typically used for exposing basic logic to end-users and programmers through weakly-typed (or non-typed) expressions. MVEL is dynamically typed (with optional typing), meaning type qualification is not required in the source, which confers significant flexibility to our purpose of creating dynamic compositional rules based on unrestricted language descriptions. For example, Listing 3 demonstrates that when user says *"search flights to Paris for less than $700"* then it is possible to dynamically create an if/then rule where some object called "flight" should have both "destination" and "price" attributes. These types are resolved at run-time using the Named-entity recognizer and the short-term memory, otherwise, if no valid resolution can be performed, then disambiguation with user will be sought. If conditions are met, object "flight" is put into the short-term (working) memory – wm. The flexibility of this approach allows us to discover and re-configure types at runtime without linking to specific classes and objects at design-time.

For Named-entity recognition, we use Stanford NER [12], a Java implementation that labels sequences of words in a text that are names of things, such as person and company names. It provides well-engineered feature extractors that annotates sentences with labels such as: NOUN, PERSON, COMPANY, NUMBER, MONEY, TIME, DATE, and LOCATION, however, since it provides a general implementation of (arbitrary order) linear chain Conditional Random Field (CRF) sequence models, it is possible to train customized models on labeled data extracted from service descriptions. In the example shown in Listing 3, NER is able to infer that “Paris” is a LOCATION, “$700” is MONEY, and “flight” is a NOUN. Service Matching module outputs a set of one or more abstract services with a similarity score associated to each service. Now, suppose that the highest similarity given the user request *“search flights to Paris...”* corresponds to the method *searchForFlights()* defined by FlightReservationService, and using the method argument descriptions (specified with @ArgDesc annotation) is possible to map the method arguments to the entities recognized by NER. The short-term memory (wm) is collecting not only the partial results and inferences produced by the forward chaining process of the rule engine but also keeps updated information collected from sensors (in the case of an Android phone), user preferences, service status, and QoS features.

```
MVELRule rule = new MVELRule()
  .name("rule-search-cheap-flights")
  .description("search flights to Paris for less than 700 US")
  .when("flight.destination == 'Paris' && flight.price < 700")
  .then("wm.put(flight); ");
```

Listing 3: Excerpt for Compositional rule expressed in MVEL

F. QoS Aware Composition

Developer may define a set of QoS features for each concrete service. Using a similar approach as described in the previous section, we define a set of heuristics (rules) that are later validated using the rule engine. Every QoS has different triggering priorities, so for instance, *battery consumption* has higher priority than *connectivity* (since some services can still work locally even when there is no connectivity, but no services may work when battery is drained), which in turn has more priority than, let’s say *accuracy* (since two high-accuracy services may compete to be selected, however if they do not run locally but remotely and WiFi connection is disabled, then neither of them can be executed). After the rule engine fires the heuristics and validates the QoS features of the concrete service candidates generated by the Service Coordinator, then a new subset of pre-selected service candidates is generated and passed to the Executor.

G. Service Discovery and Execution

Service execution selects the more appropriate concrete service based on the following criteria: if the subset of concrete service candidates contains only one element, then this service is executed, otherwise, a similarity-based selection is performed as described on the pseudocode listing below. Basically, in order to be selected, a service must have the highest similarity, which should be above an upper threshold (0.6), if not, services that are in the range $(t2 \geq cs \geq t1)$ and if their similarities differ in less than delta (0.01) then they need to be disambiguated by the user, otherwise the similarity between user request and service description is too low that no service can be selected and user needs to re-phrase the request. Values for thresholds and delta have been discovered empirically and have demonstrated satisfactory results. Finally, service discovery is performed by the OSGi Felix framework based on the selected concrete service, and if the service is available, then it is executed.

```
algorithm execute is
    input: Set of concrete services CS
    output: selected concrete service cs
    t1 := 0.6 //upper threshold
    t2 := 0.2 //lower threshold
    delta := 0.01
    cs := CS[0]
    for each pair of services (s1, s2) in CS do
        sim1 := s1.similarity
        sim2 := s2.similarity
        if sim1 >= t1 and abs(sim1 - sim2) <= delta then
            cs := max(cs.similarity, sim1, sim2)
        else if sim2 >= t1
            cs := s2
        else if sim1 >= t2  or sim2 >= t2 then
            cs := user_disambiguate(s1, s2)
        else
            cs := nil //service does not exist
    end for
return cs
```

Listing 1. Pseudocode for Service Execution

IV. Evaluation

We evaluated *NLSC* from two different perspectives in order to address the initial research questions: a precision/recall analysis to measure the performance of our system in real scenarios with users, and a metric-based analysis to estimate the amount of effort (person/day) that may be minimized when using our approach vs. using a conventional approach.

A. Performance: Recall, Precision and F1-Score

Setup: for this experiment, we conducted a user study via Amazon Mechanical Turk where 20 users participated. We provided users 15 different services and 3 scenarios (plan a trip, plan a romantic dinner, and plan a party at home next weekend). Users were asked to describe what kind of requests and questions (using unrestricted natural language) they would ask to their phones in the three different scenarios. Conversations were logged and analyzed through a confusion matrix to determine the recall, precision and F1-score metrics. For this experiment, we used 2 pre-trained models, one uses 19.7B words from 700 dimensions trained on English tweets, and and the other uses 1.7B words from 700 dimensions trained on Wikipedia entries.

Results: For the Actual Class, we defined two values: 1) YES: user’s sentence is well structured, has meaning, can be understood, and should lead to the activation of a service and a specific method, and 2) NO: user’s sentence is ambiguous, or out of context, or incomprehensible, or should not lead to the activation of a service (method is not available or do not exist). For the Predicted Class, we defined two values: 1) YES: *NLSC* has correctly identified the method and service OR if user sentence was ambiguous, then it should ask to re-phrase the sentence, and 2) NO: *NLSC* selected a wrong service and method OR id did not ask user to re-phrase the sentence. Results are summarized in table I. Since we have an uneven class distribution, that is, false positives and false negatives are very different, then Accuracy metric is not of too much help, thus we need to rely on F1-score instead due to it computes the weighted average of Precision and Recall. Generally speaking, values results demonstrate a good performance since the recall, precision an F1-Score are above 0.5. However, it is worth noting that these values can vary significantly from one experiment to the other since they rely on human judgment, which is bias-prone.

Table I
CONFUSION MATRIX FOR 3 SCENARIOS AND 20 PARTICIPANTS

Metric: Confusion Matrix		**Predicted Class**	
		YES	***NO***
Actual Class	**YES**	341	109
Actual Class	**YES**	162	76
Accuracy	0.60		
Recall	0.67		
Precision	0.81		
F1-Score	0.74		

B. *Effort Estimation*

Setup: for this experiment, we performed an analysis using Funtion Points (FP), a widely accepted industry standard (ISO/IEC 20926:2009) for functional sizing. FP are units of measurement that express the amount of business functionality that an information system provides to a user. FP are estimated in terms of both data and transaction functions. As data functions, this metric estimates the amount and complexity of Internal Logical Files (ILF) and External Interface Files (EIF), and as transaction functions it estimates External Inputs (EI), External Outputs (EO) and External Inquiries (EQ). The corresponding FP's for each function have associated a complexity measure that can be Low (L), Average (A) or High (H). We compared (analytically) the effort to develop a service composition for the "plan a trip to Paris" goal using 8 different services (FlightReservation, HotelReservation, Calendar, Weather, GroundTransportation, Messaging, LeisureActivities, and Maps) and using 2 different development approaches: *NLSC* vs. a conventional service composition schema that uses WSDL templates for service descriptions, OWL-S for semantic matching, and BPEL4J for service coordination.

Results: Based on the results presented on Table II, we can observe that the main difference between both implementations was an increment of 9 FPs (for data functions) when using the conventional approach, which means that our approach reduced the amount of data functionality to be developed in 75%. On the other hand, we can observe an increment of 4 FPs (for transaction functions) when using the conventional approach, which means that our approach reduced the amount of data functionality to be developed in $\cong 45\%$. The Total Function Point measure (TFP), which represents the total number of FPs after applying both an adjustment and a calibration factor (we used a conservative low factor of 7), reflects that the whole app is $\cong 41\%$ smaller in functionality (meaning that less functionality has to be implemented to meet the same system's requirements) when using *NLSC* instead of the conventional approach, which in turns represents a drop in effort in the same proportion.

The Effort Person/Day (EPD) estimation is computed as $EPD = TFP/DR * DPM$, where DR is the delivery rate (in average, an Android developer can implement 10 FPs per month [17]) and DPM is days per person-month (21.5 business days per month). EPD can be better understood in terms of time and number of persons required to develop the app, let's say we have a team of 5 persons, using the conventional approach it would take 31.8 days (159/5) while using *NLSC* would take 17.8 days (89/5), which means a reduction of $\cong 44$ of the required effort when using our approach.

Table II
EFFORT ESTIMATION FOR *NLSC* VS. CONVENTIONAL APPROACH

Function Points Estimation			
Metric	*NLSC*	**Conventional**	**Improvement**
ILF + EIF (FP)	3	12	75.00%
EI + EO + EQ (FP)	5	9	44.44%
TFP (FP)	23	39	41.02%
EPD (person/day)	89	159	44.02%

V. RELATED WORK

Users interact instinctively with the system in an easily expressible natural language and thus expect the system to identify the set of services that are required to achieve the user's goal. In our study, we review natural language-based approaches for dynamic service composition. If we consider an user's natural language description at one end of the problem and services at the other end, then, we find that existing literature can be broadly categorized as approaches that a) apply restrictions on how the user expresses the goal using sentence templates and/or user utterances and then use structured parsing techniques to parse the sentences to match against service names and descriptions [4] [23]; b) construct semantic graphs that represent the service description [13] such that those could be matched with the natural language descriptions using a lexical database such as WordNet, that groups words based on their meanings, to calculate a conceptual distance metric between concepts at both ends, [26] [9]; and c) match partially-observable natural language description with that of the semantics of the service described using semantic web services such as OWL-S and VDL [27] [8]. Categorical limitations of existing approaches include, (i) complex linguistic processing that employs several NLP techniques: structured parsing, extracting parts-of-speech tokens, stop-word removal, spell-checking, stemming, and text segmentation, (ii) inclusion of lexical databases such as WordNet or domain-specific ontologies that represents domain lexicons, and (iii) a weaker concept representation and similarity score for semantic matching that does not account for sentence context. To overcome the above limitations, in our work, we (a) allow users to express their sentences template-free and use their natural language description without complex linguistic processing by aligning it with service descriptions using Sentence embeddings, (b) avoid the need for lexical databases and ontologies by relying on the automatically extracted corpora of service descriptions which would otherwise be provided by service developers as code comments on services; this reduces the need to construct semantic graphs of concepts and domain-specific ontologies, and (c) use a stronger representation of words, concepts and natural language sentences that account for word usage in context to user's sentence by applying a state-of-the-art pre-trained semantic representation model of English language.

VI. Conclusions and Future Work

In this paper we have presented *NLSC*, a Dynamic Service Composition Middleware based on unrestricted Natural Language service descriptions. Using our approach, we have demonstrated that total effort (in terms of person/day) for service composition and service integration can be dramatically reduced up to 44% thanks to we eliminated the Translation phase proposed by the Service Composition Middleware (SCM) model and substituted it by an intuitive mechanism for service description, discovery, and retrieval. We also demonstrated that Service Composition using Sentence Embeddings and Named Entity Recognition techniques alleviate the burdensome task of writing boilerplate code, strictly defining well-defined hard-typed interfaces, validating ontology models and representations, and creating ad-hoc semantic reasoning mechanisms for service matching

Future Work: we plan to improve the precision of our model by training custom service description models in addition to common-sense pre-trained models as Wikipedia or Twitter entries. Also, we plan to extend our approach so it can discover third-party services published in well-known public repositories such as ProgrammableWeb.com and GitHub.

Discussion: data-driven ML and NLP approaches raise several open questions including learning with limited data. For instance, a) learning QoS-aware models that introduce model sparsity, b) inferring custom entities using reinforcement and online learning, with initial disambiguations by user, to improve service matches, c) learning context-sensitive models with working memory for better entity resolution, and d) one-shot learning from descriptions for service disambiguation.

References

[1] O. Alliance. (2018, Aug.) The dynamic module system for java. [Online]. Available: https://www.osgi.org/

[2] Apache. (2015, Nov.) Apache felix framework. [Online]. Available: http://felix.apache.org/documentation/subprojects/apache-felix-framework/apache-felix-framework-and-google-android.html

[3] S. Balzer and T. Liebig, "Bridging the gap between abstract and concrete services a semantic approach for grounding owl-s," in *Semantic Web Services:Preparing to Meet the World of Business Applications*, 2004.

[4] A. Bosca, F. Corno, G. Valetto, and R. Maglione, "On-the-fly construction of web services compositions from natural language requests," *JSW*, vol. 1, no. 1, pp. 40–50, 2006.

[5] S. Bouzefrane, D. Huang, and P. Paradinas, "An osgi-based service oriented architecture for android software development platf." 11 2011.

[6] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha, "Service composition for mobile environments," *Mob. Netw. Appl.*, vol. 10, pp. 435–451, 2005.

[7] T.-W. Chang, "Android/osgi-based vehicular network management system," in *2010 The 12th International Conference on Advanced Communication Technology (ICACT)*, vol. 2, Feb 2010, pp. 1644–1649.

[8] Y. Charif and N. Sabouret, "An overview of semantic web services composition approaches," *Theo. CS*, vol. 146, no. 1, pp. 33–41, 2006.

[9] M. Cremene, J. Tigli, S. Lavirotte, F. Pop, M. Riveill, and G. Rey, "Service composition based on natural language requests," in *Int. Conf. on Services Computing*, 2009, pp. 486–489.

[10] Easy-Rules. (2018, Aug.) A simple rule-based system. [Online]. Available: https://github.com/j-easy/easy-rules

[11] C. Escoffier. (2008, Oct.) ipojo on android. [Online]. Available: http://ipojo-dark-side.blogspot.de/2008/10/ipojo-on-android.html

[12] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *Computational Linguistics*, 2005, pp. 363–370.

[13] K. Fujii and T. Suda, "Semantics-based dynamic service composition," *Communications*, vol. 23, no. 12, pp. 2361–2372, 2005.

[14] E. Hadj, "A language-based approach for web service composition," Ph.D. dissertation, Universite de Bordeaux, France, Nov. 2017.

[15] N. Ibrahim, F. Le Mouël, and S. Frénot, "MySIM: A spontaneous service integration middleware for pervasive environments," in *Pervasive Services*. ACM, 2009, pp. 1–10.

[16] N. Ibrahim and F. L. Mouël, "A survey on service composition middleware in pervasive environments," *CoRR*, vol. abs/0909.2183, 2009.

[17] IFPUG. (2018, Jul.) Function point metrics. [Online]. Available: http://www.ifpug.org/isbsg/

[18] J. Kalinowski and L. Braubach, "Integrating application-oriented middleware into the android operating system," in *UBICOMM*, 2015.

[19] R. Karunamurthy, F. Khendek, and R. H. Glitho, "A novel architecture for web service composition," *NCA*, vol. 35, no. 2, pp. 787–802, 2012.

[20] C. Lee, S. Ko, S. Lee, W. Lee, and S. Helal, "Context-aware service composition for mobile network environments," in *Ubiquitous Intelligence and Computing*, 2007, pp. 941–952.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, p. 3111–3119.

[22] MVEL. (2018, May) Mvel guide. [Online]. Available: https://github.com/imona/tutorial/wiki/MVEL-Guide

[23] A. Ordoñez, J. C. Corrales, and P. Falcarin, "Natural language processing based services composition for environmental management," in *System of Systems Engineering (SoSE)*, 2012, pp. 497–502.

[24] M. Pagliardini and P. Gupta, "Unsupervised learning of sentence embeddings using compositional n-gram features," in *Com. Ling,*, 2018.

[25] I. Paik, W. Chen, and M. N. Huhns, "A scalable architecture for automatic service composition," *TSC*, vol. 7, no. 1, pp. 82–95, 2014.

[26] F. Pop, M. Cremene, M. Vaida, and M. Riveill, "On-demand service composition based on natural language requests," in *Wireless On-Demand Network Systems and Services*, 2009, pp. 45–48.

[27] F.-C. Pop, M. Cremene, M. Vaida, and M. Riveill, "Natural language service composition with request disambiguation," in *SOCo*, P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds., 2010, pp. 670–677.

[28] H. Pourreza and P. Graham, "On the fly service composition for local interaction environments," in *Pervasive Computing and Communications*, 2006, pp. 6 pp.–399.

[29] R. P. D. Redondo, A. F. Vilas, M. R. Cabrer, J. J. P. Arias, and M. R. Lopez, "Enhancing residential gateways: Osgi services composition," in *Conf. on Consumer Electronics*, 2007, pp. 1–2.

[30] E. Sirin, B. Parsia, D. Wu, and J. Hendler, "HTN planning for web service composition using shop2," *WS*, vol. 1, pp. 377–396, 2004.

[31] T. G. Stavropoulos, D. Vrakas, and I. Vlahavas, "A survey of service composition in ambient intelligence environments," *Artificial Intelligence Review*, vol. 40, no. 3, pp. 247–270, Oct 2013.

[32] K. Tari, Y. Amirat, A. Chibani, A. Yachir, and A. Mellouk, "Context-aware dynamic service composition in ubiquitous environment," *IEEE International Conference on Communications*, pp. 1–6, 2010.