Composing Web Services Using a Multi-Agent Framework

Yu Zhao, Daniel Alencar da Costa, *Member, IEEE* and Ying Zou, *Senior Member, IEEE*

Abstract—Different web services can be composed to perform increasingly complex tasks (e.g., making an on-line payment). However, existing approaches compose web services with hard-coded control and data flows. To proactively and autonomously compose web services, developers can develop agents. However, the development of agents for service composition is complex, due to the reasons that: 1) developers may not have the knowledge from various domains to identify the necessary tasks to carry out the required web services; and 2) a deep understanding of the agent specific code is required in order to implement agents. To alleviate the required efforts to develop agents, we propose an approach to separate the development of agent specific code from the business logic code in the service composition. More specifically, we provide an easy-to-understand syntax that abstracts agent specific code and automatically generates executable agent code. Our experimental results show that our approach can accurately identify tasks for service composition with an Area Under the Curve (AUC) of 0.88. Our experiments also demonstrate that our approach can correctly generate agent code from seven agent specifications. Finally, our user studies reveal that developers are satisfied with our approach to develop agents for service composition.

Index Terms—service composition, multi-agent framework, Jadex, multilayer perceptron model, semi-natural language syntax

1 Introduction

Web services are employed as a significant part of software systems in various domains, such as e-commerce or internet banking. In such software systems, a set of logically related web services are usually composed together [1] (*i.e.*, composite services) to fulfill complex tasks. For example, a composite service can combine a web service for searching products, with a web service for handling payments, to provide a shopping service. However, web services can be only executed once called upon. Therefore, traditional approaches compose web services with hard-coded control and data flows, which limit the reusability and maintainability of composite services [2].

To complement the traditional service composition, we adopt the agent paradigm in the service composition. The agent paradigm is an architectural style to develop software entities, *i.e.*, *agents*, which exhibit the goal-directed behavior to proactively perform actions to achieve goals [3]. There are four main advantages to apply the agent paradigm for software systems, *i.e.*, proactiveness, autonomy, reactiveness and social ability [3] [4]. We envision that the agent paradigm can also provide the four advantages for service composition: 1) proactiveness, which allows an agent to take the initiative to execute web services to satisfy their assigned goals; 2) autonomy, *i.e.*, the failure of a web service may trigger an agent to select alternative web services; 3) reactiveness, which endows the ability of an agent to constantly monitor context changes (*e.g.*, the price of a product) and trigger web services correspondingly; and 4) social ability, *i.e.*, agents are capable of cooperating with each other to perform composite services. The cooperation can be realized using asynchronous messages. Therefore, composite services are not enforced with hard-coded control and data flows.

Notwithstanding the powerful capabilities provided by agents for service composition, software developers are still required to possess a deep knowledge of the specific domain to successfully compose web services. For example, an agent designed to buy a laptop needs to implement a task to place the order for the laptop. However, a task to explore discount codes for university students could also be very imporant (since students may have a tight budget). Developers who are unfamiliar with the education domain may easily ignore such a task [5]. Without deeply understanding the target domain of an information system, the composite services may have competitive disadvantage [6]. Moroever, existing studies have shown that designing and implementing agents are complex due to the proactive, autonomous, reactive and social characteristics of agents [7] [8]. In order to use agents to improve the flexibility in service composition, developers need to invest a substantial effort to learn the agent programming due to the complexity of the agent programming [9].

To ease developers to apply the agent paradigm in service composition, we propose a semi-natural language approach that allows developers to specify high level functionalities of agents in a semi-natural language syntax. The specifications are then automatically converted into executable agent code. Our approach can help developers focus on the domain logic of the application instead of being concerned with the agent specific code.

Research has been invested considerably in *Domain Specific Languages* (DSL) to generate agent code. Several approaches transform the design of agents into agent code [7],

- *Yu Zhao is with the Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada. E-mail: yu.zhao@queensu.ca*
- *Ying Zou is with the Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada. E-mail: ying.zou@queensu.ca*
- *Daniel Alencar da Costa is with the Department of Information Science, University of Otago, Dunedin, New Zealand. E-mail: danielcalencar@otago.ac.nz*

[8], [10]. For example, the DSLs developed by Bergenti *et al.* [8] and SEA_ML [7], [10] can be used to generate agent code specific to the JADE and Jadex platforms, respectively. Hahn *et al.* [11] apply two model transformations to transform the designed metamodel to agent code used in the JACK and JADE platforms. Freitas *et al.* [12] model agents as ontologies that can generate skeleton code. Although the aforementioned approaches provide DLSs to generate agent code, these approaches have not been applied for *service composition*. Therefore, developers still need to write agent specific code that is unrelated to the domain logic of the application. Our proposed approach, on the contrary, provides built-in constructs to ease the adoption of agent based technologies in service composition.

Our approach is generic and can be applied for any agent-programming platforms. In this paper, we use the Jadex platform [13] to validate our approach. Although other agent platforms (such as, Jason [14], JACK [15], 3APL [16] and BDI4JADE [17]) can be used to develop agents for service composition, we choose the Jadex platform because Jadex agents support the use of web services. More specifically, Jadex agents use Jadex services for communication. Jadex services can be published as web services, which have a standard communication interface that supports the integration of Jadex agents with existing software applications [18].

We evaluate our proposed approach through empirical studies and user studies. We design seven agent specifications using our approach for three practical service composition scenarios. Our obtained results are promising and we list them below:

- Our approach applies the multilayer perceptron (MLP) model [19], a deep learning based model, to identify web service related tasks from on-line how-to instructions for service composition. The identified tasks enrich developers' knowledge of designing agents for service composition.
- We propose an abstract semi-natural language syntax for developers to specify functionalities of agents. The proposed syntax guides developers to implement customized code to generate executable agent code for service composition.
- Our results show that our MLP model achieves an Area Under the Curve (AUC) of 0.88 for identifying web service related tasks. Our approach can correctly generate executable agent code from the designed seven agent specifications. To examine the practical usefulness of our approach, we conduct user studies by recruiting 27 developers. The results of our user studies reveal that our semi-natural language syntax is easier to understand and adopt when compared with the state-of-the-art agent programming approach in Jadex that uses Java. Moreover, our user study shows that subjects can use our approach to correctly develop agents without explicit instructions.

Paper Organization. Section 2 presents the background. Section 3 motivates to use agents for service composition. Section 4 shows an overview of our approach to generate executable agent code. Section 5 describes the empirical studies. Section 6 discusses the limitations and generality of our approach. Section 7 summarizes the related literature. Finally, Section 8 concludes our work.

2 Background

In this section, we provide the necessary background material to our readers. We explain the concepts related to agent programming using the Jadex platform and the how-to instructions that can be used to gain domain knowledge.

2.1 Agent Programming Using the Jadex Platform

In our approach, developers use the Jadex platform [13] to build agents for service composition. The Jadex platform follows the Belief-Desire-Intention (BDI) agent paradigm [20]. The BDI paradigm simulates human behaviors in terms of selecting actions to achieve goals. In the BDI paradigm, there are three essential concepts, *i.e.*, beliefs, goals and plans. Beliefs are the knowledge that an agent has about the environment and its internal state, *e.g.*, the price of a laptop. Goals are the target states that an agent tries to achieve, *e.g.*, having an order placed. A goal has associated plans that specify actions to achieve the goal. In the Jadex platform, an action may carry out business logic, perform a web service or request to achieve a set of sub-goals to accomplish a goal. The sub-goals are achieved by executing their associated plans. Agents use Jadex services to communicate with each other. A Jadex service can be used to receive messages from other agents, process and return the responses.

Agents are programmed using Java in the Jadex platform. We classify the agent code in Jadex into two types, *i.e.*, *agent specific code* and *business logic code*. The *agent specific code* is a standard code template that implements beliefs, goals, plans and Jadex services, while the *business logic code* is the code customized by developers for achieving business logic. Figure 1 shows an example of agent code for sending messages between two agents. As illustrated in Figure 1, the *agent specific code* in grey accounts for a large portion of the agent programming in Jadex. In the example, the agent sends a laptop model to the *getBrand* Jadex service provided by the *PlaceOrder* agent, and receives the brand of a laptop.

2.2 How-to Instructions

Figure 2a shows an example of a how-to instruction for buying a laptop.[1] Typically, a how-to instruction consists of a title and a list of steps. The title describes the intention of the instruction, while each step represents a goal that needs to be achieved to fulfill the intention of the instruction. Each step has a name and a textual description of the tasks. These tasks may contain **web service related tasks**, *i.e.*, the tasks that can be performed by web services.

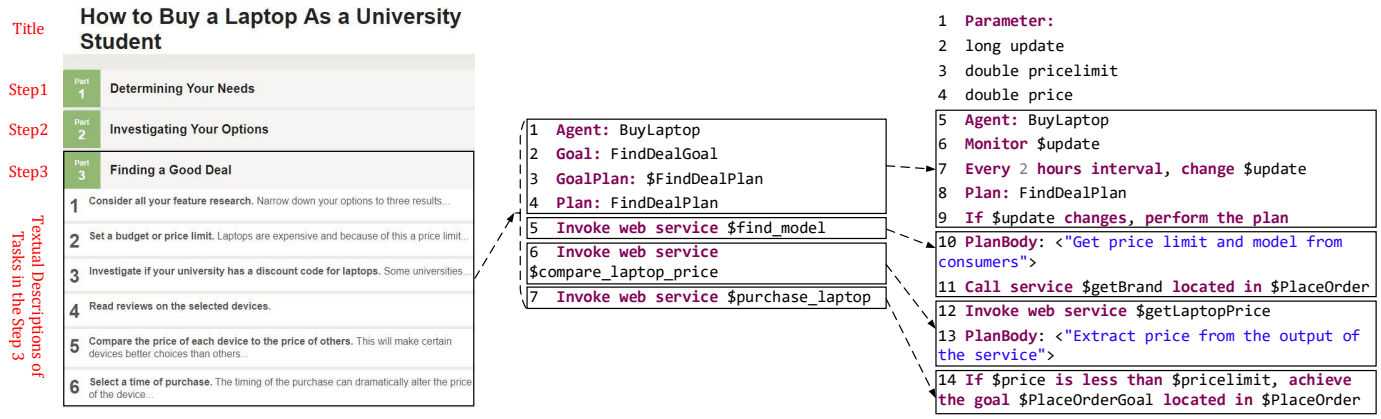
3 Motivation for Using Agents to Compose Web Services

In this section, we justify our choice for applying the agent paradigm in service composition. We discuss the advantages

[1] https://www.wikihow.com/Buy-a-Laptop-As-a-University-Student

```
final String[] brandArray = new String[1];
SServiceProvider.getServices(agent, IPlaceOrderService.class, RequiredServiceInfo.SCOPE_PLATFORM)
.addResultListener(new IntermediateDefaultResultListener<IPlaceOrderService>(){
    public void intermediateResultAvailable(IPlaceOrderService is){
        is.getBrand(model).addResultListener(new IResultListener<String>(){
            public void resultAvailable(String brand){
                brandArray[0] = brand;}
            public void exceptionOccurred(Exception exception){
                exception.printStackTrace();}});
    }
});
String returned_brand = brandArray[0];
```

Fig. 1: An example of a code snippet for two agents to exchange messages. The red code is the *business logic code*.



(a) An annotated screenshot of a how-to instruction

(b) A generated agent specification from the how-to instruction in (a)

(c) A modified agent specification from the generated agent specification in (b)

Fig. 2: An example of a how-to instruction with two examples of agent specifications for buying a laptop.

of using the agent paradigm with respect to the existing service composition approaches: *synchronous service composition* and *asynchronous service composition* [21].

Synchronous service composition follows the predefined control flows and data flows to invoke web services. The composite services are commonly modeled using *business processes* to ensure synchronized communication among web services. The *business processes* can be mapped to Business Process Execution Language (BPEL) [2] or Camunda[3] to execute composite services. Because of the predefined control and data flows, synchronous service composition can be easily implemented [22]. However, synchronous service composition has the following limitations:

- *Tight-coupling*. Web services highly depend on each other, since a web service needs to wait for the calling of the preceding web service in the control flow. Tight-coupling makes replacing individual web services difficult [2].
- *Difficulty in maintaining composite services*. The change of the interface of a web service may involve the modification of data flows among dependent web services. Accordingly, all of the composite services that involve the web service need to be adapted.
- *Difficulty in handling failures.* Web services may change dynamically because of the complex application environments [23]. For example, the description of web services can be modified and web services may be unavailable. Thus, web services may have unexpected running failures [24]. A failure of a web service results in the failures of all succeeding web services.

Asynchronous service composition does not impose procedural control flows among web services. Web services communicate with each other by exchanging messages [2]. Compared to synchronous composite services, an asynchronous composite service removes the need to wait for responses from web services. A web service reacts upon the received messages, rather than requests from specific web services in the control flow. Therefore, web services are loosely coupled. Loose coupling allows stakeholders to independently develop, deploy, update and scale their business logic to invoke web services [25]. Despite the prevalence in industry, there are limited research and public projects on asynchronous service composition [26], possibly due to the following reasons.

- *Requiring various techniques to compose web services.* Typically, to develop an asynchronous composite service, developers need a framework, (*e.g.*, Dubbo[4]) to facilitate the programming of asynchronous communication, and employ containers (*e.g.*, Docker[5]) to encapsulate the business logic to invoke web services [25]. Various techniques increase the complexity to develop composite services.
- *Lack of high-level description models.* Developers use general purpose programming languages (*e.g.*, Java) to develop asynchronous composite services. General purpose programming languages lack high-level description models

[2]https://www.w3.org/TR/2001/NOTE-wsdl-20010315
[3]https://camunda.com/
[4]http://dubbo.io/
[5]https://www.docker.com/

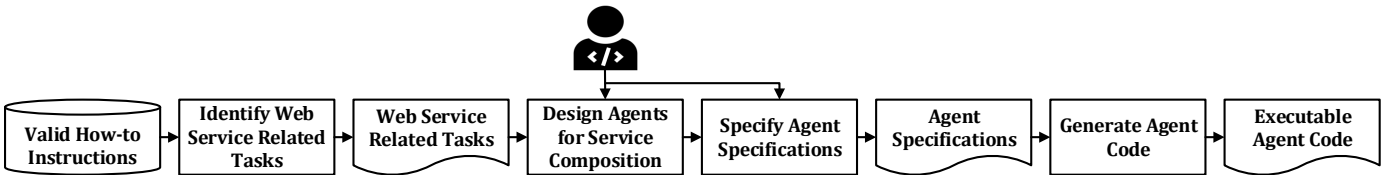


Fig. 3: An overview of our approach

to enable modularized development of composite services, which are essential for service composition [27].

- *Not autonomous.* The development of containers requires developers' knowledge to specify business logic code to invoke web services. The customized business logic code might lack the ability to proactively perform web services and autonomously decide which web services to perform.

To resolve the above mentioned limitations, agents can be designed to compose web services. Agents exist in a distributed software system, where an individual agent can act independently to perform web services. Additionally, a collection of agents can cooperate with each other to compose web services. The use of agents has the following advantages for service composition:

1) Agents support synchronous and asynchronous service composition. Agents can be synchronized to follow business processes to compose web services [28]. Moreover, the communication among agents can be asynchronous. Thus, agents can be loosely coupled and the composed web services are not imposed with procedural control flows.

2) Agents are proactive. Agents proactively perform plans to achieve the assigned goals. Plans describe business logic to invoke web services. If the interface of a web service is changed, only the business logic of the plans that involve the web service needs to be modified. Therefore, agents can cooperate with each other without knowing the internal business logic.

3) Agents are autonomous. Agents are autonomous because they can decide which web services to perform in order to achieve their goals [3]. At runtime, if the execution of a web service incurs unexpected errors (*e.g.*, the web service becomes unavailable), an agent can choose alternative web services to achieve the same goal. Therefore, although agents are statically designed at design time, the contained web services are composed dynamically at runtime.

4 Overview of Our Approach

Figure 3 presents an overview of our approach that generates executable agent code for performing service composition. Our approach consists of three main steps: 1) identifying web service related tasks from a how-to instruction; 2) designing agents for service composition based on the identified tasks; and 3) specifying agent specifications to generate executable agent code. In the following subsections, we describe each step of our approach in more detail.

4.1 Identifying Web Service Related Tasks

To relieve developers from the manual effort of studying the tasks required for service composition, we automatically identify the web service related tasks from how-to instructions. We describe our approach to identify web service related tasks in the following.

On-line how-to websites, *e.g.*, wikiHow,[6] provide how-to instructions, which are written by domain experts in natural language to teach people how to conduct diverse daily activities, such as booking a restaurant reservation or composing a song. However, how-to instructions are prepared to cover a broad range of topics. Therefore, many tasks specified in how-to instructions are manual activities and unrelated to web services, such as "inspect laptop damage".[7] To accurately filter out the tasks that are unrelated to web services, we build a multilayer perceptron (MLP) model [19], a deep learning based model, to predict whether a task described in a how-to instruction is related to web services or not.

To build the MLP model, we use the approach proposed by Zhao *et al.* [5], which analyzes grammatical structures of sentences written in natural language to extract tasks. We use the approach to extract a task (*e.g.*, "plan Disney World trip Orlando") for each sentence in a how-to instruction.

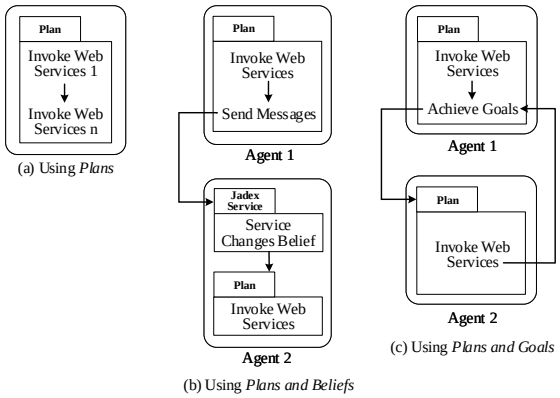
The MLP model has training and testing phases. In the training phase, a task t_i contains a learning feature vector X_i and a label l_i. Our approaches to build learning feature vectors and label tasks are described in the following steps:

1) Building learning feature vectors. We use word vectors that are pre-trained on the GoogleNews corpus by word2vec [29] to build the learning feature vectors X_i. $Word2vec$ converts a word w_i to the corresponding word vector v_i with m dimensions. Similarly to the work by Zheng *et al.* [30], we set the dimension m of a word vector v_j as 300 (*i.e.*, $v_j \in \mathcal{R}^{300}$). We map the sequence of words $(w_1, w_2, ...w_n)$ in the task to their corresponding word vectors $(v_1, v_2, ...v_n)$. For example, each word w_j in the task "plan Disney World trip Orlando" is mapped to a word vector $v_j \in \mathcal{R}^{300}$. We then average the word vectors as the feature vector $X_i \in \mathcal{R}^{300}$ for the task t_i.

2) Labeling tasks. A label l_i denotes whether a task is a web service related task or not. A l_i = 1 denotes that a task is a web service related task, while a l_i = 0 denotes, otherwise. To label a task, we need to examine if there exist web services that can perform the task. We collected a total of 4,152 web services (see Section 5.1). Manually examining such a large number of web services would be very time-consuming. Therefore, we build a Vector Space Model [31], an unsupervised ranking model, to help us label a task. The Vector Space Model measures the textual similarity between web services and a task. We use the Vector Space Model to rank the five most similar web services for the task t_i. Afterwards, we perform a manual analysis to label if the

[6]https://www.wikihow.com/

[7]https://www.wikihow.com/Buy-Used-Laptops



(a) Using *Plans*

(b) Using *Plans and Beliefs*

(c) Using *Plans and Goals*

Fig. 4: The three approaches to compose web services using the semi-natural language syntax.

task t_i is a web service related task or not. The sampling of tasks and our manual labeling process are described in Section 5.3, **RQ1**. If at least one of the ranked web services can perform the task t_i, the task is labeled as 1. Otherwise, the task is labeled as 0. Our results show that our Vector Space Model can successfully identify 81% of web service related tasks from a sample of 377 extracted tasks, which are randomly sampled with 95% confidence level and 5% confidence interval. The high accuracy indicates that we can rely on our Vector Space Model to label tasks.

In the testing phase, the MLP model takes the feature vector, X_i, of a task t_i, in the testing set as the input and predicts a label, l_i, to the task t_i. Our MLP model filters out tasks that are unrelated to web services. The filtered out tasks may contain manual activities that serve as preconditions for performing web service related tasks. For example, the filtered out manual task *set price limit* can be implemented before the web service related task to place an order for a laptop (see Figure 2c). If developers consider such filtered out tasks important, they can include the filtered out tasks to design agents.

4.2 Designing Agents for Service Composition

Given the identified tasks, developers can develop agents for service composition. In agent-oriented software engineering (AOSE), the development of agents can be divided into four phases, *i.e.*, *requirement analysis, architecture design, detailed design* and *implementation* [32]. Our work focuses on developing the internal behaviors (*e.g.*, plans) of agents. The *architecture design* phase defines the global architecture of agents to model the interactions among agents [32], which is not the focus of our work. We discuss the *architecture design* phase in Section 6. In the following, we describe the *requirement analysis* phase.

In the *requirement analysis* [32] phase, developers design the roles of agents, and the basic interactions among agents based on the selected tasks. A role describes the goals of an agent and the plans to achieve the goals at an abstract level using a natural language or UML diagrams. The designed roles serve as templates to guide the implementation of agents (described in Section 4.3). The basic interactions among agents for service composition can be designed with the following three approaches, as illustrated in Figure 4.

1) Plans (shown in Figure 4a). Multiple web services can be composed in a plan. Output messages from a web service may be served as input messages to invoke another web service in a plan. Therefore, a plan would have multiple composed web services.

2) Plans and Beliefs (shown in Figure 4b). A plan in an agent (*e.g.*, *agent 1* in Figure 4b) is performed to achieve the goal of the agent. In the plan, the output of a web service is a message that may be sent to another agent (*e.g.*, *agent 2* in Figure 4b). *Agent 1* can achieve the goal without waiting for the responses from *agent 2*. In *agent 2*, the Jadex service may change a belief value from the received message. The change of the belief value can trigger a plan, thereby invoking the web services contained in the plan.

3) Plans and Goals (shown in Figure 4c). The output from a web service in a plan can be used as a condition or an input to a request for achieving a sub-goal, as shown in *agent 1* of Figure 4c. To successfully perform the plan in *agent 1*, the agent may need to wait for the responses from the agent (*i.e.*, *agent 2*) that achieves the sub-goal. The sub-goal may be achieved by executing web services contained in the associated plan in *agent 2*.

4.3 Specifying Agent Specifications

From the designed agent roles and interactions, the internal behaviors of agents can be developed in the *detailed design* and the *implementation* phases. In the *detailed design* phase, we propose an abstracted semi-natural language syntax, a syntax that is similar to natural language, to enable modularized development of agents. The *implementation* phase concerns with programming agents. Our proposed syntax can automatically generate *agent specific code* and guide developers to implement the customized *business logic code*.

To aid developers to specify agent specifications using our proposed semi-natural language syntax, our approach automatically generates agent specifications from the identified web service related tasks in Section 4.1. In this subsection, we describe: 1) an overview of our semi-natural language syntax; and 2) our approach to automatically generate agent specifications.

4.3.1 An Overview of Our Semi-Natural Language Syntax

We use Xtext,[8] a widely used framework for building domain-specific languages (DSLs), to build our semi-natural language syntax. Our proposed syntax contains four essential components for describing an agent, including beliefs, goals, Jadex services and plans (see Section 2.1). Our proposed syntax generates executable agents that can run on the Jadex platform. Figure 2c shows an example of an agent specification for buying a laptop. The agent specification checks the price of a laptop in every 2 hours interval. If the price is lower than a price limit, the agent purchases the laptop (*i.e.*, places an order). The agent specification of the *PlaceOrder* agent in Figure 2c is shown in our online appendix.[9] We briefly describe our approach to specify each of the four components below.

[8]https://www.eclipse.org/Xtext/

[9]https://drive.google.com/open?id=1b-njDLl9udeWv64ZXERQpZtF_C27z3R-

Specifying Beliefs. The changes of belief values can be perceived by agents to trigger plans, thus invoking the contained web services. We define the semi-natural language syntax to specify beliefs. The lines 1-7 in Figure 2c give examples of our defined syntax for beliefs. In terms of programming languages, beliefs are implemented as variables (or parameters). For example, *Monitor $update* sets the parameter *update* as a belief. In our approach, a belief that is related to time can be changed to the current timestamp at certain time intervals (*e.g.*, every 2 hours). The constant changes can be used to trigger plans repetitively, e.g., to check price repetitively. We make available in our online appendix the detailed explanations for our proposed syntax.[9]

Specifying Goals. Goals are achieved by executing the associated plans, thus invoking the related web services. We define the following four syntaxes to specify a goal: (1) The *Goal Name* syntax (*i.e.*, *Goal: goal_name*) is used to declare a goal (*e.g.*, *Goal: FindDealGoal* in line 2 of Figure 2b); (2) The *Goal Input* syntax (*i.e.*, *GoalInput: $input_name*) declares input messages for a goal; (3) The *Goal Output* syntax (*i.e.*, *GoalOutput: $output_name*) declares the returned results after achieving a goal. In our approach, the above mentioned three syntaxes generate a Java class to represent a goal; and (4) The *Associated Plan* syntax (*i.e.*, *GoalPlan: $plan_name*) declares an associated plan for a goal. When a generated agent is assigned to achieve a goal, the agent takes the initiative to actively perform an associated plan of the goal, which shows the proactive characteristic of an agent [3]. A developer can declare multiple associated plans to achieve a goal in an agent specification. Thus, the autonomous characteristic of the generated agent can be reflected at run time, in which a failure of a web service could incur the agent to perform alternative plans.

Specifying Jadex Services. Jadex services receive messages from agents. Such messages can be served as conditions to perform web services, or served as inputs to the web services in a plan. To alleviate the effort of developers to manually implement Jadex services, we define the following five syntaxes to specify Jadex services: (1) The *Service Name* syntax (*i.e.*, *Service: service_name*) declares a Jadex service (*e.g.*, *Service: getBrand*); (2) The *Service Input* syntax (*i.e.*, *ServiceInput: $input_name*) declares input messages for a Jadex service; (3) The *Service Output* syntax (*i.e.*, *ServiceOutput: $output_name*) describes the returned messages of a Jadex service; (4) The *Service Changes Belief* syntax (*i.e.*, *The service changes $Beliefname*) generates a routine code to set a belief value (*i.e.*, *$Beliefname*) with the received message (see Figure 4b). The change of the belief value may trigger plans to perform web services; and (5) The *Service Body* syntax (*i.e.*, *ServiceBody: <STRING>*) allows developers to specify a customized Jadex service implementation.

Specifying Plans. A plan may activate the contained web services. Multiple agents can cooperate to perform plans, thus composing web services. A plan contains three components, *i.e.*, a plan name, plan conditions and a plan body.

To allow developers to specify plans, we define three categories of syntax: *Plan Name*, *Plan Condition* and *Plan Body*. Table 1 demonstrates our syntax to specify plans using the example *FindDealPlan* plan (see Figure 2c lines 8-14). In Table 1, the fifth column shows the generated agent code from the examples. The *Plan Name* syntax determines the name of a plan. In the following, we discuss 1) the *Plan Condition* syntax and 2) the *Plan Body* syntax.

1) **Plan Condition** specifies under which conditions a plan should be executed (*i.e.*, the *Plan Pre-condition* syntax) or dropped (*i.e.*, the *Plan Context Condition* syntax). The conditions are specified using the *ConditionLanguage* syntax. The *ConditionLanguage* syntax is shown in Table 4 of the appendix.[9] We allow developers to specify 6 categories of conditions, including *Change, Less than, Greater than, Target, Boolean* and *Contain*. The *Plan Condition* syntax endows the reactive characteristic of generated agents, in which agents check belief values with the specified conditions to decide to trigger or drop plans.

2) **Plan Body** contains *agent specific code* and *business logic code* to specify actions. The *business logic code* includes the code to invoke web services. We describe our syntax to specify the *Plan Body* below.

The *Plan Body Language* syntax allows developers to insert *business logic code*. The code is developed in Java.

Achieve Goals provides a syntax for a plan to request an agent to achieve the declared sub-goals (see Figure 4c). The requested agent may execute the associated plans of the sub-goals. Therefore, web services contained in the plans are composed. There are two syntaxes for achieving sub-goals: (1) *Achieve Goals (Same)* denotes that the sub-goal and the plan that requests to achieve the sub-goal belong to the same agent; and (2) *Achieve Goals (Diff)* indicates that the sub-goal and the plan belong to two different agents. For example, the *FindDealPlan* plan (*i.e.*, line 14 in Figure 2c) requests to achieve the sub-goal *PlaceOrderGoal* that is declared in the *PlaceOrder* agent. Using the *Achieve Goals (Diff)* syntax, the generated agents cooperate with each other to achieve a goal, which reflects the social characteristic of agents.

The *Send Messages* syntax allows an agent to call the provided Jadex services of another agent to send messages. The sent messages are specified by developers.

Invoke Web Services provides a syntax to assist developers in identifying web services and generating the code to invoke web services. Our approach uses web services that are described in the OpenAPI specification, a human and machine readable web service description language.[10] To identify web services, we use the task that is specified in the *Invoke Web Services* syntax (*i.e.*, *task_name* in Table 1). We use the Vector Space Model as described in Section 4.1 to rank the five most similar web services for the task. A developer then selects a web service that can perform the task. From the OpenAPI specification of the selected web service, our approach generates the client libraries to consume the web service. Developers can specify customized code to invoke the web service using the generated libraries.

4.3.2 Generating Agent Specifications

Our approach transforms how-to instructions to agent specifications that are specified with our proposed semi-natural language syntax. Figure 2b shows the generated agent specification for the *Finding a Good Deal* step of the how-to

[10]https://www.openapis.org/

TABLE 1: The semi-natural language syntax for specifying plans. The blue keywords represent the syntax constructs.

		Semi-Natural Syntax	Example	Agent Code
	Plan Name	Plan: plan_name	Plan: FindDealPlan	protected void FindDealPlan(){}
Plan Condition	**Plan Pre-condition**	(If|When) ConditionLanguage, (perform|apply) (the|its) plan	If $update changes, perform the plan	@Plan(trigger=@Trigger (factchangeds="update"))
	Plan Context Condition	(If|When) ConditionLanguage, (stop performing|aborting) (the|its) plan	If $context becomes false, aborting the plan	@PlanContextCondition (beliefs="context") public boolean checkCondition(){ return context!=false;}
Plan Body	**Plan Body Language**	PlanBody: <STRING>	<"Extract price from the output of the service">	*customized code for plan body*
	Achieve Goals (Same)	((If|When) ConditionLanguage,)? (Achieve|achieve) the goal $goal_name	If $price is less than $pricelimit, achieve the goal $PlaceOrderGoal	if (price <pricelimit){ final boolean success = (boolean)bdiFeature. dispatchTopLevelGoal (new PlaceOrderGoal (model)).get();}
	Achieve Goals (Diff)	((If|When) ConditionLanguage,)? (Achieve|achieve) the goal $goal_name (located|available) in $agent_name	If $price is less than $pricelimit, achieve the goal $PlaceOrderGoal located in $PlaceOrder	
	Send Messages	(Invoke|Call) service $service_name (located|available) in $agent_name	Call service $getBrand located in $PlaceOrder	See Figure 1
	Invoke Web Services	(Invoke|Call) web service $task_name	Invoke web service $PlaceLaptopOrder	*customized code to invoke web services*

instruction shown in Figure 2a. We describe our approach to generate agent specifications below.

As various how-to instructions describe distinct intentions, our approach transforms each how-to instruction to an agent specification. The title of a how-to instruction is the name of an agent. A step of a how-to instruction may contain a goal and web service related tasks that describe the plan to achieve the goal. If all of the tasks in a step are not web service related tasks, the step is identified as irrelevant to compose web services, and is filtered out.

The step name (*e.g.*, *Finding a Good Deal*) is the name of the corresponding goal (*e.g.*, *Goal: FindDealGoal*). A goal has an associated plan to achieve the goal. We use the step name as the name of the associated plan (*e.g.*, *Plan: FindDealPlan*). The goal is linked with the plan using the *Associated Plan* syntax (*e.g.*, *GoalPlan: $FindDealPlan*). The plan body consists of a number of *Invoke Web Services* syntaxes, depending on the number of web service related tasks that are identified from the step. Each identified web service related task is transformed to the *Invoke Web Services* syntax. For example, from the *Finding a Good Deal* step, our built MLP model identifies three web service related tasks, *i.e.*, *find model*, *compare laptop price* and *purchase laptop*. The three tasks are transformed to three *Invoke Web Services* syntaxes (*e.g.*, *Invoke Web Service $find_model*), respectively. The generated agent specification serves as a starting point for developers to further design the agent specification.

4.4 Generating Agent Code

Our approach traverses the syntax described in the agent specification to generate executable agent code. The example of the generated agent code from Figure 2c is included in our on-line appendix.[9] Many syntaxes (*e.g.*, the *Goal Name* syntax) can automatically generate the *agent specific code*. Other syntaxes (*e.g.*, the *Invoke Web Services* syntax) allow developers to specify the customized *business logic code* to form a plan body or a service body. The *business logic code* is saved to a Java method. To extract the *business logic code* that is implemented by developers, we analyze the Abstract Syntax Tree (AST) [18] of a Java method to parse and extract the code. A plan body or a service body is formed by concatenating the extracted *business logic code* and the generated *agent specific code*. Moreover, the necessary class headers and import statements are automatically generated based on the declared components in the agent specification.

5 Experiments

In this section, we present the setup of our experiments to evaluate our approach and the obtained results.

5.1 Data Collection

Collecting Web Services. To identify web service related tasks and compose web services using our multi-agent framework, we collect web services that are described in the OpenAPI specification. The web services are collected from two sources, *i.e.*, SwaggerHub[11] and APIs.guru[12] on December 4th, 2018. *SwaggerHub* is a platform to design and index web services. Our approach collects *published* web services in the platform. An *unpublished* web service cannot be consumed, while a *published* web service is ready

[11] https://app.swaggerhub.com/home
[12] https://apis.guru/browse-apis/

to be consumed. In total, we collect 3,140 web services from *SwaggerHub*. *APIs.guru* is an open-source project that collects publicly available web services. In total, we collect 1,012 web services from *APIs.guru*.

Collecting How-to Instructions. We collect 2,279 how-to instructions from the *wikiHow* website. The collected how-to instructions fall into 20 categories that are listed in the *wikiHow* website,[6] *e.g.*, *Arts and Entertainment*, *Computers and Electronics*, and *Education and Communications*.

5.2 Experiment Setup

Prototype Tool. We implement a prototype tool of our approach for developers to develop composite services using the proposed semi-natural language syntax. Our tool is a plugin that can be used in popular Integrated Development Environments (*i.e.*, IDEs), such as Eclipse.[13] **User Study.** To evaluate developers' satisfaction when using our approach to program agents for service composition, we setup two user studies that are described as follows.

User Study 1. The user study compares our approach with the Java programming approach, *i.e.*, the agent programming approach in Jadex. In the literature, many abstract languages are proposed to enable model-driven agent programming [8] [10] [12]. However, to the best of our knowledge, none of the languages are designed for using the multi-agent framework to develop executable composite services. In the current state-of-the-art of the Jadex platform, developers use Java to develop agents. Thus, we compare our approach with the Java programming approach. In *User Study 1*, the subjects follow our provided tutorial[14] to develop agents for service composition using both of the two approaches, *i.e.*, our semi-natural language approach and the Java programming approach. The subjects are not aware of which approach is ours in the user study. After developing agents using the two approaches, we ask the subjects to fill a survey[15] to evaluate the two approaches.

User Study 2. To evaluate our approach in a realistic scenario, the subjects use our approach to develop agents for an additional composite service. In *User Study 2*, we have not provided explicit instructions on how to write the agent code line by line. Instead, the subjects specify agent specifications and develop customized *business logic code*. The *business logic code* may include the code to invoke web services. We provide web services that are described in the OpenAPI specification. The subjects can refer to the tutorial that is used in *User Study 1* to have the knowledge of our approach for agent programming. The subjects fill a survey[16] after the user study.

To allow the subjects to program agents in the two user studies, we build a standalone Eclipse application that includes our prototype tool and Java Development Kit (JDK). Our built application is published on-line.[14] We draw two $50 Amazon gift cards for the subjects as a compensation for taking part in the user study. The user study is approved from the General Research Ethics Board (GREB) for ethical compliance. We sent emails to 42 candidates to recruit subjects. In total, we have recruited 27 subjects. Out of the 27 subjects, 48% of subjects (13/27) are professional software developers, and the other 52% of subjects (14/27) are graduate students. 22 subjects agree to take *User Study 1*, and 14 subjects agree to take *User Study 2*. The subjects do not have background on agent programming, but have at least two years of Java experience. Our experiments answer the following research questions.

RQ1. Can our approach effectively identify web service related tasks?

RQ2. Can our approach correctly generate agent code?

RQ3. Are developers satisfied with our approach to program agents?

5.3 Research Questions

RQ1. Can our approach effectively identify web service related tasks?

Motivation. To relieve developers from the effort of designing composite services using the multi-agent framework, we propose an approach that automatically identifies web service related tasks from how-to instructions. We provide guidelines for developers to specify agent specifications using the identified tasks. We are interested to evaluate the performance of our approach on identifying web service related tasks.

Approach. A large number of our collected how-to instructions, e.g., "*How to Cook Dumplings*", are clearly invalid for service composition as they describe manual steps. Therefore, we filter out invalid how-to instructions that are not relevant to service composition. Our approach to filter out invalid how-to instructions is described in the online appendix.[9] In total, we verified and collected 275 how-to instructions that are relevant for service composition.

From the collected valid how-to instructions, we extracted 21,441 tasks using the approach proposed by Zhao *et al.* [5]. To build the MLP model (as described in Section 4.1), we randomly sampled 2,680 tasks (*i.e.*, 12.5% of the extracted tasks). The first two co-authors independently performed a manual analysis to label whether the sampled tasks were web service related tasks or not. Our manual labeling results showed that we achieved the Cohen's kappa [33] value of 0.64 between the two co-authors, indicating a substantial agreement. The two co-authors had a discussion session to reach a consensus on the disagreement. To train and test the performance of the proposed MLP model, we randomly splitted the sampled tasks into training, validation and testing sets using a ratio of 80:10:10. We used the validation set to tune the hyper-parameters of the MLP model. Different settings of hyper-parameters could influence the performance of the proposed model. Thus, we empirically examined the effect of two hyper-parameters: 1) the number of hidden layers and 2) the number of neurons.

We evaluate the performance of our MLP model using *Precision*, *Recall*, *F-Measure* and *Area Under the Curve (AUC)*.

- *Precision* measures the ratio of correctly predicted web service related tasks from the set of tasks that are predicted as related to web services.

[13] https://www.eclipse.org/

[14] https://drive.google.com/open?id=1tp8mTTzqqfqgIDaqm5J4XJ6a_sIBudCs

[15] https://forms.gle/WmcUbjTxD26G6ysW9

[16] https://forms.gle/yGU8wZjcTbjzGbA69

TABLE 2: The performance comparison of our model with three baseline models on the testing set.

Model	Precision	Recall	F-Measure	AUC
MLP	0.67	0.56	**0.61**	**0.88**
Logistic Regression	0.74	0.43	0.54	0.88
Random Forest	**0.86**	0.13	0.22	0.79
Naïve Bayes Classifier	0.29	**0.60**	0.39	0.71

- *Recall* calculates the fraction of the web service related tasks that our approach could retrieve.
- *F-Measure* computes the harmonic average of the *Precision* and *Recall* to quantify the accuracy of our prediction results.
- *Area Under the Curve (AUC)* measures the area under the curve that plots the true positive rates against the false positive rates [34]. The values of AUC range from 0.5 to 1. A value of 0.5 represents random guessing, while 1 denotes that all the web service related tasks can be properly distinguished.

To test the effectiveness of our models, we compare our model with the following three baselines, *i.e.*, *Logistic Regression*, *Random Forest* and *Naïve Bayes Classifier*.

- *Logistic Regression* is a well-known binary classification model that combines learning features linearly.
- *Random Forest* is an ensemble of decision trees to perform classification or regression tasks. The prediction results consider each individual decision tree, thus preventing over-fitting.
- *Naïve Bayes Classifier* is a simple probabilistic model for classification. The model assumes the conditional independence among the learning features.

The three baseline models and our model are built with the same learning features and labels.

Results. Our approach can effectively identify web service related tasks. Our proposed MLP model obtains the best validation performance on *F-Measure* and *AUC* when there is 1 hidden layer and 8 neurons in the hidden layer. Thus, we empirically set the number of hidden layer as 1 and the number of neurons as 8 in our MLP model. Table 2 shows the performance comparison between our model and the three baseline models. Overall, our model achieves the best *F-Measure* (*i.e.*, 0.61) and *AUC* (*i.e.*, 0.88). AUC shows true capability of a model to distinguish between web service related tasks and other tasks, because AUC is not dependent on arbitrary threshold. Compared with our model, the *Logistic Regression* model achieves the same AUC value, but decreases the *Recall* by 13%. The lower *Recall* value of the *Logistic Regression* model produces lower *F-Measure* values. These observations suggest that our MLP model has a better prediction accuracy compared with the three baseline models. Our results show that, on average, 18% of extracted tasks are web service related tasks. The results indicate that the manual identification has a maximum precision of 18%. In contrast, our approach achieves a precision of 67%, which improves the efficiency of developers to manually identify tasks by nearly four times. Moreover, our approach can identify a majority of web service related tasks (*i.e.*, 56%).

RQ2. Can our approach correctly generate agent code?

Motivation. Our approach generates the executable agent code using the proposed syntax. Correctly generating agent code demonstrates the applicability of our approach. In this research question, we exploit the proactive, autonomous, reactive and social characteristics to design agents and evaluate the correctness of the generated agent code.

Approach. We apply our approach to run three real service composition scenarios, *i.e.*, *Going Shopping*, *Checking Book Descriptions* and *Purchasing a Pet*. For each scenario, we use real-world web services collected from Section 5.2 and use the *Invoke Web Services* syntax to generate the code for invoking web services. To test the correctness of our approach to generate the agent code, we design seven agent specifications for the three scenarios that exercise the three approaches for service composition (*i.e.*, using *Plans*; *Plans and Beliefs*; and *Plans and Goals* defined in Section 4.2). The seven agent specifications exploit the proactive, autonomous, reactive and social characteristics, and use all the semi-natural language syntax defined in Section 4.3.1. We describe the three scenarios and the corresponding agent specifications as follows.

1) Going Shopping. A *CheckWeather* goal has a plan that uses a weather forecasting web service to check if it is raining in a specified city and set the *isRaining* belief. If *isRaining* is false, a *CheckOpeningHour* plan is then triggered. The *CheckOpeningHour* plan uses a web service to check the opening hours of a local shop. We implement two agent specifications, *i.e.*, *Case 1* and *Case 2*, for the scenario.

- *Case 1* (using *Plans and Beliefs* in one agent). The *CheckWeather* goal and the *isRaining* belief are implemented in the same agent.
- *Case 2* (using *Plans and Beliefs* in two agents). The *CheckWeather* goal and the *isRaining* belief are implemented in two agents, as shown in Figure 4b.

2) Checking Book Descriptions. A *GetBookInfo* goal returns the book description given a book name. The goal can be achieved using two plans. *Plan 1* composes two web services. A web service returns the unique ISBN number given a book name, and the other web service takes the ISBN number as an input and returns the book description. *Plan 2* invokes an atomic web service that takes the book name as an input and returns the book description. We implement two agent specifications, *i.e.*, *Case 3* and *Case 4*, for the scenario.

- *Case 3* (using *Plans* in one agent). The *GetBookInfo* goal is associated with *Plan 1*.
- *Case 4* (using *Plans* in one agents). The *GetBookInfo* goal is associated with *Plan 1* and *Plan 2*.

TABLE 3: The exploited agent characteristics.

Agent	Proactive	Autonomous	Reactive	Social
Case 1	✓		✓	
Case 2	✓		✓	✓
Case 3	✓			
Case 4	✓	✓		
Case 5	✓			
Case 6	✓			
Case 7	✓			✓

TABLE 4: The results of *User Study 1*. *Number*: number of subjects (22 in total) who prefer our approach (*i.e.*, subjects choose 4 or 5 in questions); *P*: *Number* divided by the total number of subjects; *Students*: percentage of students (13 in total) who prefer our approach; *Pros*: percentage of professional software developers (9 in total) who prefer our approach.

#	Questions	Number	P(%)	Students(%)	Pros(%)
1	To what degree do you think the syntax provided in the semi-natural language approach is easy-to-understand?	20	91%	92%	89%
2	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to understand?	19	86%	85%	89%
3	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to learn?	20	91%	92%	89%
4	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to program agents?	19	86%	92%	78%
5	Comparing the Java programming approach and the semi-natural language approach for the agent programming, which one do you think is easier to inspect, verify and maintain the code?	13	59%	62%	56%
6	Do you prefer to program agents using the Java programming approach or using the semi-natural language approach?	15	68%	69%	67%

3) Purchasing a Pet. In this scenario, a web service checks the availability of a pet. If the pet is available (*i.e.*, the pet is not sold), another web service is used to place an order. The input to the composite service is a pet ID, and the output is the order status. We implement the scenario with the following three agent specifications.

- *Case 5* (using *Plans* in one agent). The agent composes the two web services in a single plan by using two *Invoke Web Services* syntaxes in the plan body.
- *Case 6* (using *Plans and Goals* in one agent). We implement three goals in the same agent to compose web services using the *Achieve Goals (Same)* syntax.
- *Case 7* (using *Plans and Goals* in three agents). We implement three goals in three different agents that are coordinated using the *Achieve Goals (Diff)* syntax.

Our designed seven agent specifications exploit various agent characteristics, as summarized in Table 3. Agents show proactiveness when actively performing plans to achieve goals. The agent in *Case 4* has autonomy, since the agent can choose an alternative plan when the performed web service has unexpected runtime failures. Agents in *Case 1* and *Case 2* are reactive to the belief changes to perform plans. Finally, *Case 2* and *Case 7* reflect social ability, since multiple agents cooperate to compose web services.

Results. Our approach correctly generates executable agent code from the seven agent specifications. We deploy the generated agent code in the Jadex platform. We consider that the code is correctly generated if the agents generate the expected outputs from composite services. Results show that the deployed agent code correctly generates outputs. For *Case 1* and *Case 2* in the *Going Shopping* scenario, a false *isRaining* value outputs the opening hours, while a true *isRaining* value terminates the plan. *Case 3* and *Case 4* return the book description given a book name. Specifically, in *Case 4*, the web service in *Plan 2* has a runtime failure that the book name is not found. Thus the agent performs *Plan 1* to get the book description, which shows the autonomous characteristic of agents. The three agents in the *Purchasing a Pet* scenario return the order status.

Moreover, we calculate the average lines of code written in our approach and the average lines of Java code that is generated from our approach, for the seven agent specifications. We do not count the comment lines and empty lines. Results show that the lines of Java code generated from our approach (*i.e.*, 155 lines) are 3.5 times more than the lines of code written in our approach (*i.e.*, 44 lines). Our approach requires less coding, since our approach abstracts the *agent specific code* to a concise semi-natural language syntax. The *agent specific code* accounts for 78% of the generated Java code, indicating that our syntax can generate a large portion of agent code.

RQ3. Are developers satisfied with our approach to program agents?

Motivation. To evaluate whether Java developers without agent programming knowledge can easily adopt our proposed semi-natural language approach to develop agents, we conduct two user studies to investigate the perceived usefulness of our approach according to the subjects.

Approach. In *User Study 1*, the subjects follow our tutorial[14] to program the *Going Shopping* scenario with *Case 1* (described in Section 5.3 **RQ2**) using both of the Java programming approach and our semi-natural language approach. In the tutorial, we follow the description of the Jadex documentation[17] to describe the steps to perform the Java programming approach. In *User Study 1*, the code that is implemented in the Java programming approach by the subjects and the code that is generated from our semi-natural language approach are the same and executable. The subjects are able to see the outputs, *i.e.*, weather and opening hours. The survey[15] contains six single choice questions, as shown in Table 4. A question has a Likert-scale from 1 to 5 to be chosen. A scale 5 represents that the subjects understand our syntax (questions 1-2) or prefer our semi-

[17]https://download.actoron.com/docs/releases/latest/jadex-mkdocs/

natural language approach (questions 3-7). Moreover, we provide open-ended questions for the subjects to explain their choices.

In *User Study 2*, the subjects program the *Checking Book Descriptions* scenario with *Case 3* (described in Section 5.3 **RQ2**) using our semi-natural language approach. Same as *User Study 1*, the survey[16] in *User Study 2* contains six single choice questions, with a Likert-scale from 1 to 5 to be chosen. Although there are 9 subjects taking both of the user studies, our obtained results are unlikely biased, because in *User Study 2*, the subjects need to have the knowledge of our approach from *User Study 1* to develop agents.

Results. 1) *User Study 1*: Developers prefer to use our approach to develop agents. Table 4 shows the results of *User Study 1*. Overall, 91% of subjects believe our syntax is easy-to-understand. More than 86% of subjects believe that our semi-natural language approach is easier to understand, learn and program agents, compared with the Java programming approach. It is interesting to note that the percentage of students who favor our approach to program agents (92%) is slightly higher than the percentage of professional software developers (78%). The reason is probably because that professional software developers are more experienced in Java. We conclude that our syntax is understandable. Our approach saves the required efforts from developers to program agents and learn to program agents. 59% of subjects consider our approach is easier to inspect, verify and maintain the code, compared with the Java programming approach. The 59% of subjects prefer our approach because 1) the agent specification described with our proposed syntax provides high-level abstractions and logics of agents; 2) they can focus on debugging the customized code rather than the generated agent specific code; and 3) the code written in our approach is shorter. The other subjects favor the Java programming approach, because 1) the subjects are familiar with Java; and 2) our approach cannot inspect Java compilation errors of the implemented customized *business logic code*. In summary, 68% of subjects prefer to use our approach when compared with the Java programming approach. The other 23% of subjects equally like the two approaches.

2) *User Study 2*: Developers are satisfied with our approach to develop agents. Our results in *User Study 2* show that 86% (12/14) of subjects successfully programmed the agent and are able to see the output of the composite service by running the agent. On average, the subjects spent 44 minutes to program the agent from scratch. Although the subjects do not have any knowledge on agent programming, by reviewing the programming example in *User Study 1*, a majority of subjects can correctly develop agents for service composition. 79% (11/14) of subjects believe that using our approach is easy to connect the automatically generated code and the customized *business logic code* to form the executable agent code. More than 86% (12/14) of subjects believe that our approach is easy-to-understand, easy-to-learn, and easy-to-program. Overall, all of the subjects (14/14) are satisfied with our approach to develop agents. The results demonstrate that our approach reduces the efforts from developers to learn and program agents for service composition.

In summary, our semi-natural language approach separates the agent design from implementation. Developers can focus on implementing the customized *business logic code* rather than learning the agent programming domain knowledge. The automatic code generation from the designed agent specifications requires less coding efforts.

6 Discussion

In this section, we discuss the limitations and generality of our proposed approach.

Vector Space Model. Our approach uses the Vector Space Model to recommend web services. Many other approaches have been proposed to recommend web services for tasks [35] [36]. Designing novel approaches for web service recommendation is not the focus of our work. We use the Vector Space Model, because the model has been proven effective for information retrieval [5] [37], and can be easily implemented.

How-to Instructions. Our approach of identifying web service related tasks from on-line how-to instructions is general, and can be used in other agent platforms to design agents for service composition.

Semi-Natural Language Syntax. Our proposed semi-natural language syntax provides high-level abstractions to specify beliefs, goals, plans and Jadex services that are essential to describe agents for service composition. The Jadex platform uses Jadex services to enable the communication among agents. Other agent platforms, *e.g.*, Jason and JACK, use agent communication protocols for agents to communicate with each other. Although our syntax for specifying Jadex services is specific to the Jadex platform, the other syntaxes for specifying beliefs, goals and plans are general to agent platforms. Thus, our designed syntax can be transformed to other agent programming languages, *e.g.*, AgentSpeak in Jason.

Architecture Design Phase. In Section 4.2, we introduce three basic interaction approaches for agents to compose web services. In complex scenarios of agent interactions, interaction protocols as proposed by existing research [22] [38] should be defined in the *architecture design* phase. For synchronous communication, developers can use BPEL to specify procedural control flows among agents [28]. For asynchronous communication, developers can use commitment-based protocols [38] [39] and Blindingly Simple Protocol Language (BSPL) [22] [28] to develop distributed systems. As a complement to the traditional synchronous service composition (see Section 3), building a multi-agent system using asynchronous communication allows various stakeholders to compose web services to achieve a common goal. Other interaction protocols [40] [41] can be used for agents to negotiate and understand each others' capabilities and responsibilities to achieve goals. The used interaction protocols can be implemented in the *detailed design* and *implementation* phases by specifying customized business logic code in Jadex services and plans (see Section 4.3).

Generated Agent Code. Our proposed semi-natural language syntax generates agent code that is specific to the Jadex platform [13]. The generated agent code from our proposed syntax follows the standard principle of the agent programming as described in the Jadex documentation[17].

It indicates that our generated code is not more complex compared to the *agent specific code* that is hand-written with Jadex. Our approach concatenates the customized code implemented by developers and our generated code to form a plan body or a service body, as described in Section 4.4. To make the plan body or service body executable, developers need to make sure that the customized code is functionally correct and satisfies the desired business logic. Nevertheless, developers can always modify the customized code.

Types of Web Services. Our proposed *Invoke Web Services* syntax (described in Section 4.3.1) eases the effort from developers to identify and consume web services that are described with the OpenAPI specification. For web services that are described with other types of specifications, *e.g.*, WSDL and Hydra,[18] developers can use the *Plan Body Language* syntax (described in Section 4.3.1) to implement customized code to consume the web services. Our approach does not increase the complexity for consuming the web services, since the customized code is also required in the Java programming approach of Jadex.

User Study. In our user studies, the subjects develop agents for the *Going Shopping* and the *Checking Book Descriptions* scenarios (described in **RQ3**). Implementing more service composition scenarios may test the full capabilities of our approach. However, it requires substantial efforts from the subjects to perform the user study, which may discourage the participation of the subjects. We choose the two scenarios, since the developed agents in the two scenarios include four essential components of agents, *i.e.*, beliefs, goals, plans and Jadex services. In total, we recruit 27 subjects in our user studies. Although our results show that a majority of the subjects prefer our approach to program agents, involving more subjects is desirable to make our conclusions stronger.

7 Related Work

In this section, we summarize the related work on service composition, the integration between agents and web services, and the agent-oriented software engineering.

Service composition is a process to combine a set of logically related web services. Extensive research efforts have focused on automatically composing web services to satisfy the specified user constraints and preferences. For example, some approaches [1] [42] [43] use AI-planning algorithms (*e.g.*, reinforcement learning) to match predefined tasks with web services. Other approaches find all of the possible solutions of composite web services by matching the semantic relations between input and output parameters of web services [44] [45] [46]. The above mentioned automatic composite services cannot be directly deployed in real scenarios due to the required complex business logic for connecting web services [47]. In contrast, we propose a programming model to develop executable composite services using the multi-agent framework.

Several programming models are proposed to develop executable composite services. For example, the mashup technology [48] provides graphical user interfaces (GUI) to deploy and link web services in a drag-and-drop manner. Other approaches use business process engines to compose web services [24] [49] [50]. For example, Afzal *et al.* [50] help developers to map web services with tasks that are specified in the existing business processes. Liu *et al.* [51] propose a DSL to develop and deploy user interfaces into BPEL engines for service composition. Lee *et al.* [52] design adapters on BPEL engines to support the composition of heterogeneous web services (*e.g.*, RESTful web services). The above mentioned approaches ensure the synchronization of web services to follow rigid control flows and data flows. In contrast, agents as used in our approach have the advantages of proactiveness, autonomy, reactiveness and social ability to support synchronous and asynchronous composition of web services. In asynchronous service composition, web services are loosely coupled.

The integration between agents and web services is studied for decades. WSIGS [53] and WS2JADE [54] focus on designing a gateway to enable the translation between agent messages and web service messages. The gateway has a heavy burden to manage the communication among multiple agents and web services [55]. Casals *et al.* [56] apply a relaying gateway to publish agents as web services. Our Jadex platform natively supports the publication of agents as web services. Other approaches facilitate the dynamic cooperation between agents and web services. For example, Corchado *et al.* [57] and SEA_ML [7] [10] use agents as controllers to discover and communicate with web services dynamically at runtime. However, it is difficult to guarantee that the discovered web services meet the functional and non-functional requirements. In contrast, our approach supports developers to statically compose web services using the agent paradigm at design time. In multi-agent systems, *artifacts* are proposed as non-autonomous environment objects that can be used by agents to achieve goals [58]. Specifically, CArtAgO-WS [55] uses artifacts to access web services. Different from CArtAgO-WS, our approach uses plans that describe business logic to invoke web services. While we focus on developing internal behaviors of agents, CArtAgO-WS models the environment objects.

Agent-oriented software engineering (AOSE) provides high-level abstractions to support the development of agents. We summarize the related work on AOSE, since we propose the abstracted semi-natural language syntax to develop agents for service composition. Our agent development approach focuses on the *detailed design* phase and the *implementation* phase.

The *detailed design* phase uses abstracted modeling language to model internal behaviors of agents. For example, Bergenti *et al.* [8] propose a DSL with grammers extended from Xtend[19] (a language similar to Java), to model agents in the JADE platform. Tezel *et al.* [59] develop a metamodel that is composed of graphical notations for modeling agents in the Jason platform. SEA_ML [7] [10] designs a DSL to model agents in the Jadex platform. Other approaches [11] [60] present metamodels or DSLs that are independent of agent platforms. Same as the aforementioned approaches, our proposed approach enables modularized agent development. Differently, we propose a semi-natural language

[18] http://www.hydra-cg.com/spec/latest/core/#introduction

[19] https://www.eclipse.org/xtend/

syntax to develop agents. Our proposed syntax is easy to be understood by developers.

The *implementation* phase concerns with programming agents using specific agent platforms. Many approaches transform the design of agents in the *detailed design* phase to the agent code. For example, the DSLs developed by Bergenti *et al.* [8] and SEA_ML [7] [10] can be used to generate agent code that is specific to the JADE and Jadex platform, respectively. Hahn *et al.* [11] apply two model transformations to transform the designed metamodel to agent code that is used in the JACK and JADE platform. Freitas *et al.* [12] model agents as ontologies that can generate skeleton code. The above mentioned approaches are not designed for service composition. Using the above mentioned approaches, developers may still need to implement *agent specific code* that is related to the agent concepts. Unlike the aforementioned approaches, our approach tightly integrates the *detailed design* phase with the *implementation* phase. Thus, the semi-natural language syntax guides developers to generate executable agent code for service composition.

8 Conclusion

The mutli-agent framework can be applied to build agents and perform composite services, due to the advantages of proactiveness, autonomy, reactiveness and social ability in agents. However, developers may not be proficient in various domain knowledge when designing composite services in agents. To ease developers to apply the Jadex platform in service composition, we provide an approach to use a semi-natural language syntax to develop agents. By using our approach, developers can implement *business logic code* without the need of knowing the agent specific code. Our case study results demonstrate that we can effectively identify web service related tasks from on-line how-to instructions. Moreover, our approach can correctly generate executable agent code. The results of our user studies show that the subjects prefer to use our approach.

In future work, we plan to improve the maintainability of our approach to develop agents. We can provide a user-friendly interface for developers to easily navigate between our syntax and the corresponding customized *business logic code*. Moreover, to facilitate the *architecture design* phase, we can provide schemas for developers to specify various interaction protocols, *e.g.*, BSPL, and semi-natural language syntax to specify protocol-compliant agents.

References

[1] Y. Zhao, S. Wang, Y. Zou, J. Ng, and T. Ng, "Automatically learning user preferences for personalized service composition," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 776–783.

[2] Amazon. [Online]. Available: https://d0.awsstatic.com/whitepapers/microservices-onaws.pdf

[3] M. Wooldridge, *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[4] F. Zambonelli, N. R. Jennings, and M. Wooldridge, "Developing multiagent systems: The gaia methodology," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 3, pp. 317–370, 2003.

[5] Y. Zhao, S. Wang, Y. Zou, J. Ng, and T. Ng, "Mining user intents to compose services for end-users," in *ICWS*. IEEE, 2016, pp. 348–355.

[6] N. Zhang, J. Wang, and Y. Ma, "Mining domain knowledge on service goals from textual service descriptions," *IEEE Transactions on Services Computing*, 2017.

[7] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas, and T. Kosar, "On the use of a domain-specific modeling language in the development of multiagent systems," *Engineering Applications of Artificial Intelligence*, vol. 28, pp. 111–141, 2014.

[8] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "Agent-oriented model-driven development for jade with the jadel programming language," *Computer Languages, Systems & Structures*, vol. 50, pp. 142–158, 2017.

[9] S. Getir, M. Challenger, and G. Kardas, "The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems," *International Journal of Cooperative Information Systems*, vol. 23, no. 03, p. 1450005, 2014.

[10] M. Challenger, M. Mernik, G. Kardas, and T. Kosar, "Declarative specifications for the development of multi-agent systems," *Computer Standards & Interfaces*, vol. 43, pp. 91–115, 2016.

[11] C. Hahn, C. Madrigal-Mora, and K. Fischer, "A platform-independent metamodel for multiagent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 18, no. 2, pp. 239–266, 2009.

[12] A. Freitas, R. H. Bordini, and R. Vieira, "Designing multi-agent systems from ontology models," in *International Workshop on Engineering Multi-Agent Systems*. Springer, 2018, pp. 76–95.

[13] A. Pokahr, L. Braubach, and W. Lamersdorf, "Jadex: A bdi reasoning engine," in *Multi-agent programming*. Springer, 2005, pp. 149–174.

[14] R. H. Bordini and J. F. Hübner, "Bdi agent programming in agentspeak using jason," in *International Workshop on Computational Logic in Multi-Agent Systems*. Springer, 2005, pp. 143–164.

[15] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas, "Jack intelligent agents-summary of an agent infrastructure," in *5th International conference on autonomous agents*, 2001.

[16] M. Dastani, M. van Birna Riemsdijk, and J.-J. C. Meyer, "Programming multi-agent systems in 3apl," in *Multi-agent programming*. Springer, 2005, pp. 39–67.

[17] I. Nunes, C. Lucena, and M. Luck, "Bdi4jade: a bdi layer on top of jade," in *Proc. of the Workshop on Programming Multiagent Systems*, 2011, pp. 88–103.

[18] Y. Zhao, Y. Zou, J. Ng, and D. A. da Costa, "An automatic approach for transforming iot applications to restful services on the cloud," in *ICSOC*. Springer, 2017.

[19] S. S. Haykin *et al.*, *Neural networks and learning machines/Simon Haykin.* New York: Prentice Hall,, 2009.

[20] M. Bratman, "Intention, plans, and practical reason," 1987.

[21] W. Song and H.-A. Jacobsen, "Static and dynamic process change," *IEEE Transactions on Services Computing*, vol. 11, no. 1, pp. 215–231, 2016.

[22] M. P. Singh, "Information-driven interaction-oriented programming: Bspl, the blindingly simple protocol language," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 491–498.

[23] C. Lv, W. Jiang, S. Hu, J. Wang, G. Lu, and Z. Liu, "Efficient dynamic evolution of service composition," *IEEE Transactions on Services Computing*, vol. 11, no. 4, pp. 630–643, 2015.

[24] A. L. Schwerz, R. Liberato, C. Pu, and J. E. Ferreira, "Robust and reliable process-aware information systems," *IEEE Transactions on Services Computing*, 2018.

[25] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, 2018.

[26] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Poster: Benchmarking microservice systems for software engineering research," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 323–324.

[27] W. Song, H.-A. Jacobsen, C. Ye, and X. Ma, "Process discovery from dependence-complete event logs," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 714–727, 2015.

[28] A. Günay and A. K. Chopra, "Stellar: A programming model for developing protocol-compliant agents," in *Preproceedings of the 6th International Workshop on Engineering Multi-Agent Systems*, 2018, pp. 1–20.

[29] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their com-

positionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.
[30] L. Zheng, V. Noroozi, and P. S. Yu, "Joint deep modeling of users and items using reviews for recommendation," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 2017, pp. 425–434.
[31] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press, 2008, vol. 1, no. 1.
[32] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.
[33] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.
[34] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.
[35] C. Lin, A. Kalia, J. Xiao, M. Vukovic, and N. Anerousis, "Nl2api: A framework for bootstrapping service recommendation using natural language queries," in *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, 2018, pp. 235–242.
[36] M. Tang, Y. Jiang, J. Liu, and X. Liu, "Location-aware collaborative filtering for qos-based service recommendation," in *2012 IEEE 19th International Conference on Web Services*. IEEE, 2012, pp. 202–209.
[37] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242. Piscataway, NJ, 2003, pp. 133–142.
[38] P. Yolum and M. P. Singh, "Commitment machines," in *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 2001, pp. 235–247.
[39] M. Baldoni, C. Baroglio, and F. Capuzzimati, "A commitment-based infrastructure for programming socio-technical systems," *ACM Transactions on Internet Technology (TOIT)*, vol. 14, no. 4, p. 23, 2014.
[40] M. Baldoni, C. Baroglio, K. M. May, R. Micalizio, and S. Tedeschi, "Computational accountability in mas organizations with adopt," *Applied Sciences*, vol. 8, no. 4, p. 489, 2018.
[41] A. K. Chopra and M. P. Singh, "From social machines to social protocols: Software engineering foundations for sociotechnical systems," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 903–914.
[42] L. Ren, W. Wang, and H. Xu, "A reinforcement learning method for constraint-satisfied services composition," *IEEE Transactions on Services Computing*, 2017.
[43] A. Mostafa and M. Zhang, "Multi-objective service composition in uncertain environments," *IEEE Transactions on Services Computing*, 2015.
[44] A. S. Da Silva, H. Ma, Y. Mei, and M. Zhang, "A hybrid memetic approach for fully automated multi-objective web service composition," in *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, 2018, pp. 26–33.
[45] S. Chattopadhyay and A. Banerjee, "Qos constrained large scale web service composition using abstraction refinement," *IEEE Transactions on Services Computing*, 2017.
[46] J. Li, Y. Yan, and D. Lemire, "Full solution indexing for top-k web service composition," *IEEE Transactions on Services Computing*, vol. 11, no. 3, pp. 521–533, 2016.
[47] I. Paik, W. Chen, and M. N. Huhns, "A scalable architecture for automatic service composition," *IEEE Transactions on Services Computing*, vol. 7, no. 1, pp. 82–95, 2012.
[48] Q. Bao, J. Zhang, X. Duan, R. Ramachandran, T. J. Lee, Y. Zhang, Y. Xu, S. Lee, L. Pan, P. Gatlin *et al.*, "A fine-grained api link prediction approach supporting mashup recommendation," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 220–228.
[49] Z. Zheng, K. S. Trivedi, K. Qiu, and R. Xia, "Semi-markov models of composite web services for their performance, reliability and bottlenecks," *IEEE Transactions on Services Computing*, vol. 10, no. 3, pp. 448–460, 2015.
[50] A. Afzal, B. Shafiq, S. Shamail, A. Elahraf, J. Vaidya, and N. R. Adam, "Assemble: Attribute, structure and semantics based service mapping approach for collaborative business process development," *IEEE Transactions on Services Computing*, 2018.
[51] X.-z. Liu, M. Xu, T. Teng, G. Huang, and H. Mei, "Muit: A domain-specific language and its middleware for adaptive mobile web-based user interfaces in ws-bpel," *IEEE Transactions on Services Computing*, 2016.
[52] J. Lee, S.-J. Lee, and P.-F. Wang, "A framework for composing soap, non-soap and non-web services," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 240–250, 2014.
[53] D. Greenwood and M. Calisti, "Engineering web service-agent integration," in *SMC*, vol. 2. IEEE, 2004, pp. 1918–1925.
[54] X. T. Nguyen and R. Kowalczyk, "Ws2jade: Integrating web service with jade agents," in *International Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering*. Springer, 2007, pp. 147–159.
[55] A. Ricci, E. Denti, and M. Piunti, "A platform for developing soa/ws applications as open and heterogeneous multi-agent systems," *Multiagent and Grid Systems*, vol. 6, no. 2, pp. 105–132, 2010.
[56] A. Casals, A. E. F. Seghrouchni, O. Negroni, and A. Othmani, "Exposing agents as web services in jade," in *EMAS*, 2019.
[57] J. M. Corchado, D. I. Tapia, and J. Bajo, "A multi-agent architecture for distributed services and applications," *IJICIC*, vol. 8, no. 4, pp. 2453–2476, 2012.
[58] A. Omicini, A. Ricci, and M. Viroli, "Artifacts in the a&a meta-model for multi-agent systems," *Autonomous agents and multi-agent systems*, vol. 17, no. 3, pp. 432–456, 2008.
[59] B. T. Tezel, M. Challenger, and G. Kardas, "A metamodel for jason bdi agents," in *OASIcs-OpenAccess Series in Informatics*, vol. 51, 2016.
[60] S. Rodriguez, N. Gaud, and S. Galland, "Sarl: a general-purpose agent-oriented programming language," in *WI-IAT*, vol. 3. IEEE, 2014, pp. 103–110.

Yu Zhao is currently a researcher at Huawei Human Machine Interaction Lab, Toronto, Canada. He received the Ph.D degree from the Department of Electrical and Computer Engineering at Queen's University, Canada, in 2019. He was the recipient of the IBM Ph.D Fellowship Award. His research interests include service-oriented computing, recommender systems, Internet-of-Things and natural language understanding. More about Yu and his work is available at https://yzhao.org

Daniel Alencar da Costa is a Lecturer (Assistant Professor) at the University of Otago, New Zealand. Daniel obtained his PhD in Computer Science at the Federal University of Rio Grande do Norte (UFRN) in 2017 followed by a Postdoctoral Fellowship at Queen's University, Canada, from 2017 to late 2018. His research goal is to advance the body of knowledge of Software Engineering methodologies through empirical studies using statistical and machine learning approaches as well as consulting and documenting the experience of Software Engineering practitioners.

Ying Zou is the Canada Research Chair in Software Evolution. She is a professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture. More about Ying and her work is available online at https://www.ece.queensu.ca/people/Y-Zou/