Using Workflow Patterns to Model and Validate Service Requirements

Ye Wang, Bo Jiang
School of Computer and Information Engineering
Zhejiang Gongshang University
Hangzhou, China
{yewang, nancybjiang}@zjgsu.edu.cn

Ting Wang
College of Computer Science and Technology
Zhejiang University of Technology
Hangzhou, China
wangting@zjut.edu.cn

***Abstract*—Service requirements documentation plays a crucial role on the quality of service-oriented systems to be developed. A large amount of service requirements are documented in the form of natural language, which are usually human-centric and therefore error-prone and inaccurate. In order to improve the quality of service requirements documents, we propose a service requirements modeling and validation method using workflow patterns. First, it extracts the process information using natural language processing tools. Then it formalizes the process information with a requirements modeling language – Workflow-Patterns-based Process Language (WPPL). Finally, the defects existed in service requirements are checked against a set of checking rules by matching with workflow patterns. A financial service example – Trade Order – was used to illustrate our approach.**

***Index Terms*—Service requirements, workflow patterns, natural language processing, requirements modeling, requirements validation.**

I. Introduction

Service requirements describe the requirements for customer or market to such aspects as the functionality of services, quality of services, etc [1]. Service requirements are not easily to handle due to the following characteristics: 1) they are constantly changing with the change of user needs and expectations [2]; 2) they are process-driven [3] and therefore address more on service interactions among multiple agents [4].

The quality of service-oriented systems to be developed greatly relies on the quality of service requirements documents. Due to the fact that most service requirements are still described in natural language (NL), such human-centric NL-based requirements are error-prone, inaccurate and lack of structure. However, most research on service-oriented systems pays more attention on the service design and implementation, rather than service requirements engineering. In order to bridge the gap between service design and service requirements, this paper presents an approach to model and validate service requirements by reducing requirements defects (e.g. inconsistency, incorrectness, incompleteness) and therefore improve the quality of service requirements.

There is a lot of established work on the requirements validation [5][6][7]. Most of these approaches [8][9] use formal reasoning to model and validate requirements. However, while these heavyweight techniques are useful, it is advocated to "develop a more lightweight approach to support translation between natural language and semi-formal requirements models" [10]. As patterns have been attracting more and more attention in requirements engineering, we adopt patterns to develop our lightweight approach. More specifically, we employ workflow patterns to model the process-centric service requirements, due to the fact that workflow is one preferred way to help requirements engineers, particularly towards modeling process descriptions. We first use workflow patterns and natural language processing techniques to semi-automatically extract the process information from NL-based service requirements documents, and then use Workflow-Patterns-based Process Language (WPPL) to model the process information, which is further validated by checking against a set of checking rules through pattern matching. WPPL has been developed in the previous work [11]. The previous focus of our work was to develop the meta-model of WPPL and to validate requirements mainly in the form of use cases. The translation from use cases to WPPL is manual. In this paper we focus on service requirements and extend the previous work to support the semi-automatic translation from NL-based service requirements to WPPL.

The structure of this paper is as below: Section 2 provides an overview of the proposed approach and an introduction to workflow patterns as well as WPPL. Section 3 introduces the translation process from NL-based service requirements to WPPL specifications whereas Section 4 introduces the service requirements validation process. The whole approach is illustrated through a financial service example. Section 5 presents the related work and Section 6 draws conclusions.

II. An Overview

A. Workflow Patterns

A workflow is an enactment of a business process that is described in terms of a flow of work through an organization [12]. Workflow patterns are developed by workflow patterns initiative that aims to provide a collection of generic recurring process constructs. Due to its high reusability, workflow pat-

978-1-5090-3694-3/16 $31.00 © 2016 IEEE
DOI 10.1109/REW.2016.18

RE 2016 Workshops, Beijing, China
RePa Workshop Paper

terns have a wide applicability in both academia and industry [13].

Usually, service requirements are classified into two categories [14]: external requirements including the input, the output of services and the cooperation with other services, and internal requirements including the process flow within a service. Workflow patterns are developed for describing the process flow, thereafter we adopt workflow patterns as a basic benchmark to extract the process information between services and evaluate the quality of requirements documents.

As shown in Fig. 1, this work will employ five basic workflow patterns, which are sequence, exclusive choice, simple merge, parallel split and synchronization. We do not claim that the five patterns are complete to represent all workflows, but we can represent more complex workflows through combing them. For example, the recursion pattern can be jointly represented by the exclusive choice pattern and the simple merge pattern, while the multiple merge pattern can be jointly represented by the parallel split pattern and the simple merge pattern [15].

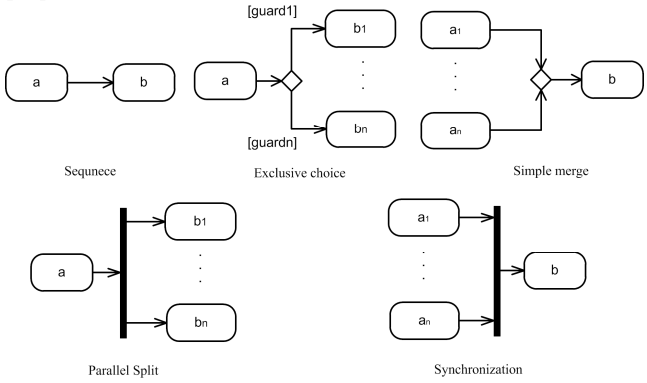


Fig. 1. UML 2.0 Activity Diagram for basic workflow patterns.

B. Workflow-Patterns-based Process Language(WPPL)

A WPPL specification describes a service requirement in the form of a business process that is composed of workflow patterns. Each service requirement has a unique *name*. Each name is defined in the action form similar to action language [16], such as *action(entity1, entity2,…)*. Action language is a language designed for modeling actions and their effects [16]. The semantics of action language are based on the assumption that "things remain the same until something happens to make them change" [17]. This assumption is the same as our assumption of representing service requirements in WPPL. Therefore we adopt action language to represent the manipulations and their manipulated objects of a service requirement.

The description of each service requirement is structured into two layers [11]. The inner layer describes the process flow within a service, whereas the outer layer depicts the key properties of a service, such as the *input*, the *output*, the *precondition* and the *postcondition*.

On the outer layer, the input data and output data are represented as *entities*. The precondition and postcondition are represented as states, in the form of *state(entity)*, which represents an entity's state. The reason why we use states to represent the precondition and postcondition is that the subsequent requirements validation process is based on the concept of IOPE (i.e., Input, Output, Precondition and Effect) [18], of which the precondition and the effect can be represented as states. The action language is used to represent the states whereas propositional logic [8] is used to represent the relationships between states. Propositions can be joined using logical connectives to make new propositions, such as logical *and*, logical *or* and so on.

On the inner layer, each service requirement can be treated as one or more business processes consisting of multiple actions connected by a sequence of control flows, which are represented by workflow patterns. More specific details can be found in the literature [11]. The Backus-Naur Form grammar of WPPL is given as follows.

/*The outer layer*/
<requirement>::= *<name> <input> <output> <precondition> <postcondition> <process>*
<name> ::= **name** *Action*
<input> ::= **input** *Entity*$^+$
<output> ::= **output** *Entity*$^+$
<precondition> ::= **precondition** *State*$^+$
<postcondition> ::= **postcondition** *State*$^+$
/*The inner layer*/
<process>::=**process** *<pattern>* | *<pattern>*;*<process>*
<pattern>::= *<pattern_name>*(*<action_set>*)
<action_set>::= *Action, Action* | *Action, <action_set>*
<pattern_name> ::= **seq** | **par_split** | **sync** | **excl_choice** | **merge**

We use *Trade Order* as an example to illustrate our approach. Trade Order is a representative service in financial trading service systems. The Trade Order service requirement is described as below:

"The trader enters a new order to the system. The operator would then match the new order with existing opposite-side orders in the order book. If the new order fails to be matched, the new order should be added to the order book; otherwise, the printing agent will print the trade to the public. After the trade is successfully printed, the system will notify the trader with the new trading information, including the traded price and trading sizes, and so on."

Fig. 2 shows a simplified example of the WPPL specification of the Trade Order service, whose name is defined as *trade(order)*. The precondition for *trade(order)* is that the initial order is prepared, i.e., *prepared(initial_order)*, whereas the postcondition is either the order is traded, i.e., *traded(order)*, or the order has been added to the order_book, i.e., *updated(order_book)*. Therefore *traded(order)* and *updated(order_book)* are joined with the logical *or*, denoted as ∨.

On the inner layer, the process contains five sub services, which might be mapped to an atomic business action, a business service or a composite service. These sub services cooperate with each other according to a set of predefined workflows. The cooperation process is represented as **seq***(enter(order), match(order));* **excl_choice***(match(order), add(order), print(trade));* **seq***(print(trade), notify(trader))*. This means that

to achieve the goal of *trade(order)*, the order needs to be entered to the system (i.e., *enter(order)*) and then be matched with opposite-side orders (i.e., *match(order)*); if no trade is generated in the process, then add the new order to order book (i.e., *add(order)*); otherwise print out the trade (i.e., *print(trade)*) and notify the trader (i.e., *notify(trader)*). In order to avoid the situation that some service requirements may share the same keywords as pattern names, abbreviations are used in WPPL to represent each pattern. For example, **seq** represents sequence, whereas **par_split** represents parallel split.

```
name trade(order)

input initial_order

output trade order_book

precondition prepared(initial_order)

postcondition traded(order) ∨updated(order_book)

process   seq(enter(order),match(order));
          excl_choice(match(order),
          add(order), print(trade));
          seq(print(trade), notify(trader))
```

Fig. 2. WPPL specification for the Trade Order service requirement.

C. Overview of Our Approach

In this paper, we propose a pattern-matching approach to model and validate service requirements, and consequently to improve the quality of service requirements documents. Fig. 3 gives an overview of our approach, which consists of the following steps:

1) Using workflow patterns to extract the inner-layer process information of WPPL specifications, including sub services and control flows between them.

2) Analyze service requirements documents to extract the outer-layer information of WPPL specifications.

3) Generate WPPL specifications by combining the results of step 1 and step 2.

4) Validate WPPL specifications by matching with workflow patterns and checking against checking rules.

5) Output the validation result.

III. Modeling Service Requirements with WPPL

The key step of transforming service requirements into WPPL specifications is to extract the inner-layer information. In order to improve the transformation efficiency, we develop an approach to semi-automatically extract the process information from service requirement documents using workflow patterns and natural language processing techniques.

The transformation process is divided into two basic steps: 1) Syntax-level analysis; 2) Text-level analysis. Syntax-level analysis aims to obtain the grammatical structure of each sentence and extract the necessary service goals from each sentence, whereas text-level analysis aims to analyze the relationship among sentences and extract the control flows from the text. The outer-layer information needs to be analyzed from service requirement documents by requirements analysts (RAs). The rest of this section will illustrate the transformation process through the Trade Order example. The structural overview of the transformation process is shown in Fig. 4.

A. Syntax-Level Analysis

As the first step of the transformation procedure, the aim of syntax-level analysis is to extract sub services, which are also treated as business services. It is worthy noting that these

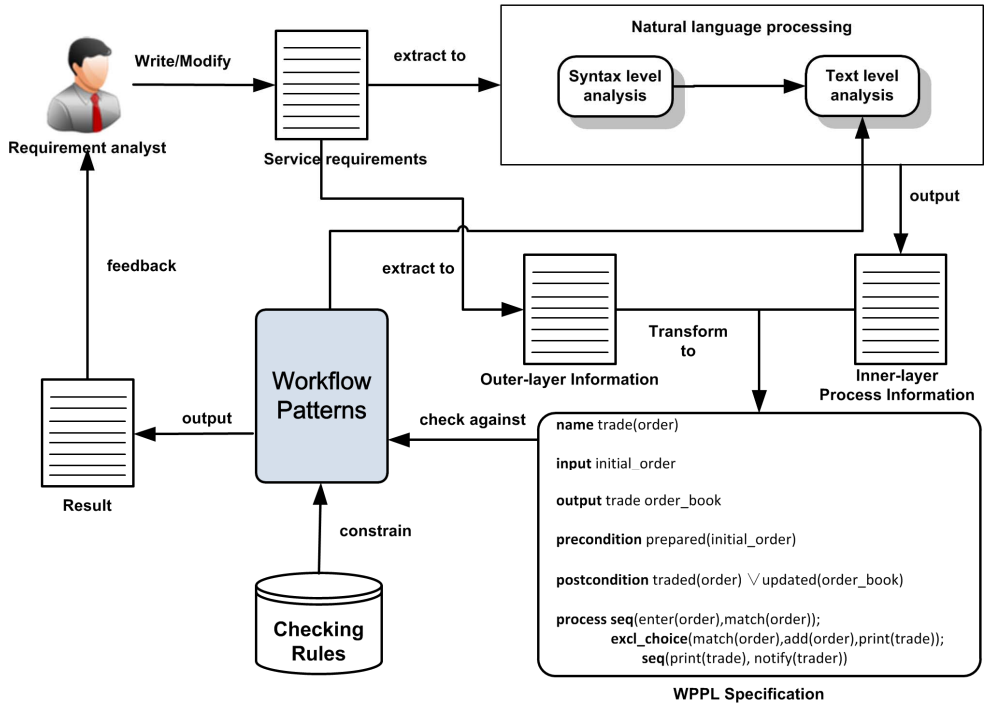


Fig. 3. Overview of our approach.

"business services" are not real service entities. They refer to the requirement goals of business services, but for simplicity, we call them business services in this paper. For example, *trade(order)* is actually a goal for the Trade Order service. As mentioned before, business services can be an action, a service or even a composite service. The natural language processing tool – Stanford Parser – is utilized to work out the grammatical structure of each sentence. It performs two main tasks: (1) identify and assign Part-of-Speech (POS) tags to the word in a sentence; (2) create grammatical relations or type dependencies among elements in a sentence, which are called as *Stanford Dependencies* (SD) [19]. Fig. 5 shows how to extract business services from service requirements documents. The process of syntax-level analysis consists of the following steps.

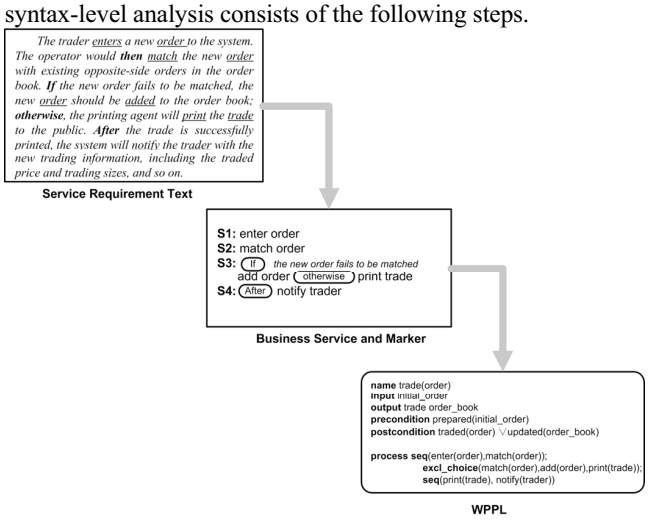


Fig. 4. The service requirements modeling process.

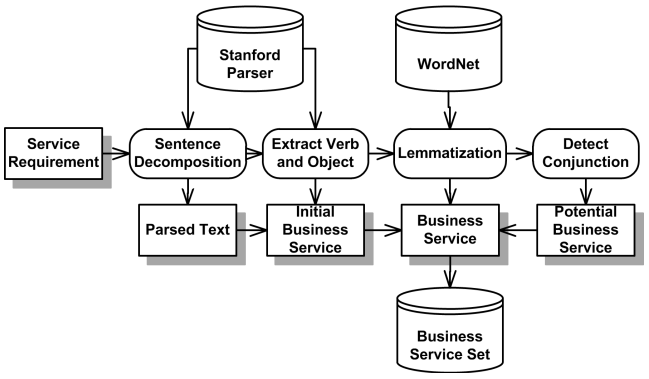


Fig. 5. Structural overview of the steps of the syntax-level analysis

Step 1: A tokenization splits up the text into individual sentences. Each sentence is parsed by Stanford Parser to obtain the SD set.

Step 2: The specific SD set will be analyzed to get Verb and Object that can comprise initial business services.

Step 3: Detect the Conjunction relationship in the SD set and find out whether there are potential business services in sentences.

Step 4: Lemmatize the words in initial business services using WordNet and generate business services finally.

Business service is generally described as a verb and a set of related parameters. The most important parameter is generally a noun or noun phrase, which is used to indicate the entity affected by the business goal. Thus a basic business service can be represented in terms of *Verb + Object*. The definition of the business service is given as below. Business service is defined as a binary relation: *(p(verb), p(object))*. The *p(verb)* is a verb or a verb phrase, which represents an executable action, whereas the *p(object)* is a noun or a noun phrase, which represents an operational object in a business service.

The *SD set* is a set of grammatical relations parsed by Stanford Parser. The current representation contains 50 grammatical relations [20], each SD is a binary relations: *SDType(governor, dependent)*, a grammatical relation holds between a *governor* and a *dependent*. For instance, relation *dobj(make, products)*, short for "direct object", means that the noun *products* is the direct object of the verb *make*. In this section, we identify four scenarios to obtain the initial business services by analyzing the SD set.

Scenario 1: Use the relation *dobj(governor, dependent)*. *dobj* defines that the direct object of a verb phrase is the noun phrase which is the accusative object of the verb [20]. In *dobj(governor, dependent)*, the governor is a verb, and the dependent is a noun or noun phrase which is the direct object of the governor. In this case, we obtain the initial business service as below.

dobj(governor, dependent) → (p(governor), p(dependent))

For instance, we can derive the SD relation *dobj(submits, order)* from the sentence "The trader submits a new order". Then the initial business service *(submits, order)* is obtained.

Scenario 2: Use the relation *nsubjpass(governor, dependent)*. *nsubjpass,* short for "passive nominal subject", defines that a passive nominal subject is a noun phrase which is the syntactic subject of a passive clause [20]. *nsubjpass(governor, dependent)* appears in the passive voice, in which the governor is a verb, and the dependent is a noun or noun phrase. In this case, we obtain the initial business service as below.

nsubjpass(governor,dependent) → (p(governor), p(dependent))

For instance, we can derive the SD relation *nsubjpass (validated, order)* from the sentence "The new order is validated by the system". Then the initial business service *(validated, order)* is obtained.

Scenario 3: Combine the relation *prep_p(governor, dependent)* with *nsubj(governor, dependent)*. *prep*, short for "prepositional modifier", defines that a prepositional modifier of a verb, adjective, or noun is any prepositional phrase that serves to modify the meaning of the verb, adjective, noun, or even another preposition [20], whereas *p* denotes the specific preposition. *nsubj*, short for "nominal subject", defines that a nominal subject is a noun phrase which is the syntactic subject of a clause [20]. When the verb part in a business service is a verb phrase contains a preposition, we can obtain the business service through the relation *prep_p*. However, not every

relation *prep_p* can extract the business service correctly. For example, for the sentence, "Bell is based in LA", the basic type dependency is *prep_in(based, LA)*. But this sentence does not imply any business service. We need to combine *prep_p* with the relation *nsubj* to facilitate more accurate business services extraction. In this case, we obtain the initial business service as follows.

IF *prep_p(governor) = nsubj(governor)*
THEN *prep_p(governor, dependent)* → *(p(governor), p(dependent))*

For instance, for the sentence, "The department will search for more music information", the SD relations *nsubj(search, department)* and *prep_for(search, information)* are derived. By combing these two relations, we obtain the initial business service *(search, information)*.

Scenario 4: Use the relation *conj(governor, dependent)* to find potential business services. *conj*, short for "conjunction", defines that the relation between two business services connected by a coordinating conjunction, such as "and", "or" [20]. In *conj(governor, dependent)*, the governor denotes the first conjunct whereas the dependent denotes other conjunctions depend on it via the *conj* relation. For example, the relation *conj_or(ski, snowboard)* is derived from the sentence "They either ski or snowboard". In this case, we obtain the potential business service as follows.

(p(verb), p(object)) ∧ *conj(p(verb), dependent)* → *(p(dependent), p(object))*
(p(verb), p(object)) ∧ *conj(governor, p(object))* → *(p(verb), p(governor))*

For instance, for the sentence "We will inspect and authorize the request", we can obtain SD relations *conj_and(inspect, authorize)* and the business service *(inspect, request)*. According to Scenario 4, we can obtain the potential business service *(authorize, request)*.

Both initial business services and potential business services need to be further processed, including deleting the word existing in the stopword list and lemmatizing words. Stop words usually refer to the most common words in a language. The business service extraction algorithm is shown as below.

Algorithm 1. The business service extraction algorithm
Input: An analyzed SD set *D* from one sentence and stopwords list *S*
Output: The business service set *N* in this sentence

1. Initialize the business service set *N*
2. **for** each *sd*∈ *D*
3. **if** *sd* can be applied to any of the four scenarios **then**
4. *bs* ← generate(*sd*) //generate the initial business service
5. *N* ← *bs* // add the initial business service to the set *N*
6. **end if**
7. **end for**
8. **for** *bs*∈ *N*
9. **if** *p(verb)*∈ *S* **then** // if the verb is a stop word
10. delete this *bs*
11. **else** lemmatize the words in the business service
12. *N* ← *bs*
13. **end for**

By parsing the aforementioned Trade Order description on the syntax level using the above algorithm, we can obtain the following business services:

{(enter, order), (match, order), (add, order), (print, trade), (notify, trader)}

B. Text-level analysis

At the syntax-level analysis stage, the cooperations among business services, i.e., the process flows or the workflows, are not taken into account. In service requirements descriptions, there are some words and phrases expressing the process flow such as "if", "otherwise", "then" or phrase "in the meanwhile", which are called *markers*. Markers or the combination of markers can be mapped to a specific basic workflow pattern. We will extract markers from service requirements documents to analyze the process flows among business services, and then generate corresponding workflows. The text-level analysis (as shown in Fig. 6) consists of the following steps.

Step 1: Extract markers that can indicate workflow patterns by analyzing the parsed text.

Step 2: Combine the markers with corresponding business services.

Step 3: Determine the workflow pattern.

Step 4: Generate the inner-layer information of WPPL

According to the five basic workflow patterns, we classify markers into four categories namely Sequence Indicator (SEQ), Condition Indicator (CON), Parallel Indicator (PAR) and Synchronization Indicator (SYN). Please note that we don't identify the merge indicator due to the fact that we don't use any explicit marker to represent the simple merge pattern. Table I lists some common markers. The markers listed in Table I do not claim completeness and can be extended by users according to different domains if necessary.

By combining adjacent business services and markers, we can determine the workflow pattern. The algorithm to determine the workflow pattern is given as follows.

Algorithm 2. The algorithm to determine the workflow pattern
Input: *BSS* (Business Service Set) and *MS* (Marker Set). *MS*

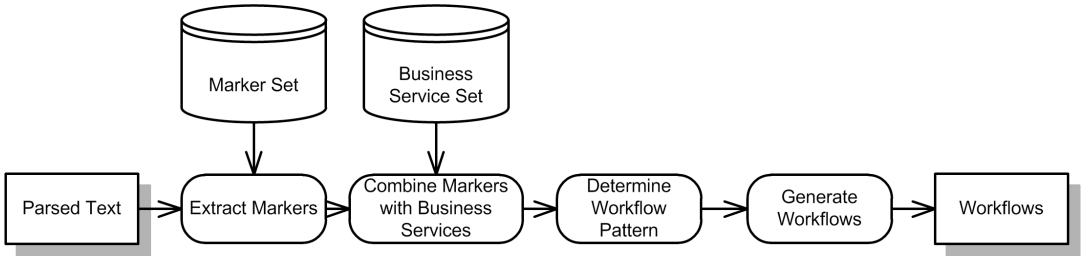


Fig. 6. Structural overview of the steps of the text-level analysis.

includes *SEQ, CON, PAR,* and *SYN*.

Output: *WPS* (Workflow Pattern Set)

1. **initialize** *nodeSplit=close*
2. **for** each *bs*∈ *BSS*
3. **if** *bs.getMarker()*∈ *SEQ* **then**
4. *WPS* ← **seq***(bs, bs.getSSeqBS())* //get the subsequent business service and add the sequence pattern to *WPS*
5. remove *bs* from *BSS*
6. **end if**
7. **if** *bs.getMarker()*∈ *CON* **then**
8. *j* = 1
9. **do** save the adjacent *bs* to the j^{th} branch and the guard condition
10. *nodeSplit* = open // record this is exclusive choice
11. **repeat** the algorithm for the j^{th} branch
12. *j++;*
13. **while**(last branch) // the exclusive choice sentence ends
14. *WPS* ← **excl_choice***(bs, bs.getExclBSs())* //get all the branching adjacent business services and add the exclusive choice pattern to *WPS*
15. remove *bs* from *BSS*
16. **end if**
17. **if** *bs.getMarker()*∈ *PAR* **then**
18. *j* = 1
19. **do** save the adjacent *bs* to the j^{th} branch
20. *nodeSplit* = open *// record this is parallel split*
21. **repeat** the algorithm for the j^{th} branch
22. *j++*
23. **while**(last branch) *// the parallel split sentence ends*
24. *WPS* ← **par_split***(bs,bs.getParBSs())* //get all the branching adjacent business services and add the parallel split pattern to *WPS*
25. remove *bs* from *BSS*
26. **end if**
27. **if** (*nodeSplit* == open && *bs.getSSeqBS()* == *bs.getAnyConBS().getSSeqBS()*) **then** //*getAnyConBS()* is to get any of the last business service of the branches except the branch of *bs*
28. *nodeSplit = close*
29. *WPS* ← **merge***(bs,bs.getConBSs(),bs.getSSeqBS())*
30. remove *bs and all bs.getConBSs()* from *BSS*
31. **end if**
32. **if** (*nodeSplit==open && bs.getMarker()*∈ *SYN*) **then**
33. *nodeSplit = close*
34. *WPS* ← **sync***(bs,bs.getSyncBSs(),bs.getSSeqBS())* //*getSyncBSs()* is to get all last business service of the branches except the branch of *bs*
35. remove *bs and all bs.getSyncBSs()* from *BSS*
36. **end if**
37. **end for**

The last step of text-level analysis is the generation of the inner-layer information of WPPL specification by connecting all business services with workflow patterns. This step needs the double check and adjustment of RAs to make sure that the workflows are correct.

After we obtain the inner-layer information of WPPL specifications, we obtain the outer-layer information of each service requriement by analyzing the documents, including its own input, output, precondition and postcondition. Fig. 2 shows the WPPL specification for the Trade Order requirement and Fig. 7 shows its UML Acitivity Diagram.

TABLE I. COMMON MARKERS

Sequence Indicator	Condition Indicator	Parallel Indicator	Synchronization Indicator
then	if	while	only if
next	otherwise	meanwhile	when
thus	else	concurrently	once
subsequently	whether	meantime	
after	optionally	in the meantime	
afterwards	either	in parallel	
and then	in case		

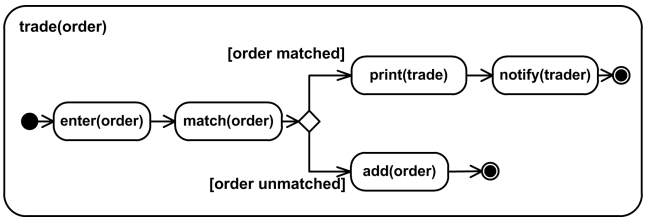


Fig. 7. UML activity diagram for Trade Order.

IV. VALIDATING SERVICE REQUIREMENTS USING WORKFLOW PATTERNS

As aforementioned, there is a large amount of inconsistency, incorrectness or incompleteness in service requirements documents. In this section, we define a set of checking rules to validate service requirements according to the five basic workflow patterns. These checking rules (see Table II), based on the concept of IOPE, are derived from the analysis experience of RAs in practice.

In order to illustrate our approach, we present another requirement document of *Trade Order* (composed of *Enter Order*, *Cross Order*, etc) by injecting some defects and transform it into WPPL specifications, as shown in Fig. 8.

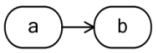
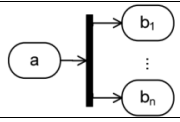
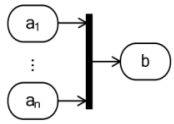
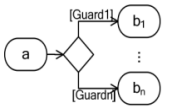
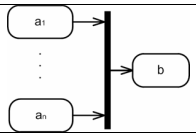
Although *Enter Order* and *Cross Order* can be decomposed into other business actions, we perform the requirements validation only at the top level for illustration due to space limitation. Therefore we only extract the outer-layer information of *Enter Order* and *Cross Order*. We can see that *Trade Order* is composed of four business services: *enter(order)*, *cross(order)*, *notify(trader)* and *add(order)*. Then we match each WPPL specification against the checking rules.

The checking result is given as follows:

1) *cross(order)* should not follow *enter(order)*, as there are no sharing states between the precondition of *cross(order)* and the postcondition of *enter(order)*;

2) *notify(trader)* is not one of the outgoing branch of *cross(order)*, as there are neither sharing states between the postcondition of *cross(order)* and the precondition of *noti-*

TABLE II. THE CHECKING RULES FOR WORKFLOW PATTERNS

Workflow Pattern	WPPL Example	Diagram	Checking Rules
Sequence	**seq**(a,b)		SEQR1:$a.output \cap b.input \neq \emptyset$ SEQR2:$a.postcondition \cap b.precondition \neq \emptyset$
Parallel Split	**par_split**($a, b_1,...,b_n$)		PSR1: $\forall\, b_i\ (i=1...n)$, $a.output \cap b_i.input \neq \emptyset$ PSR2: $\forall\, b_i\ (i=1...n)$, $a.postcondition \cap b_i.precondition \neq \emptyset$
Synchronization	**sync**($a_1,...,a_n,b$)		SYNR1: $\forall\, a_i\ (i=1...n)$, $a_i.output \cap b.input \neq \emptyset$. SYNR2: $\forall\, a_i\ (i=1...n)$, $a_i.postcondition \cap b.precondition \neq \emptyset$
Exclusive Choice	**excl_choice**($a,b_1,...,b_n$)		ECR1: $\forall\, b_i\ (i=1...n)$, $a.output \cap b_i.input \neq \emptyset$ ECR2: $\forall\, b_i\ (i=1...n)$, $a.postcondition \cap b_i.precondition \neq \emptyset$ ECR3: $\forall\, b_i,\ b_j\ (i,j=1...n)$, $i \neq j$, $b_i.precondition \neq b_j.precondition$ ECR4: $\forall\, b_i,\ b_j\ (i,j=1...n)$, $i \neq j$, $b_i.postcondition \neq b_j.postcondition$
Simple Merge	**merge**($a_1,...,a_n,b$)		SMR1: $\forall\, a_i\ (i=1...n)$, $a_i.output \cap b.input \neq \emptyset$. SMR2: $\forall\, a_i\ (i=1...n)$, $a_i.postcondition \cap b.precondition \neq \emptyset$. SMR3: $\forall\, a_i,\ a_j\ (i,j=1...n)$, $i \neq j$, $a_i.postcondition \neq a_j.postcondition$

fy(trader), nor sharing data between the output of *cross(order)* and the input of *notify(trader)*.

These defects are reported to RAs, and the reasons for the two defects have been identified. The first defect is caused by the incorrect service whereas the second defect is caused by the missing service. RAs then add *print(trade)* before *notify(trader)* and substitute *match(order)* with *cross(order)* and repeat the checking process of Trade Order until no defect can be found.

```
name trade(order)
input initial_order
output trade order_book
precondition prepared(initial_order)
postcondition traded(order) ∨updated(order_book)
process seq(enter(order),cross(order));
        excl_choice(cross(order),notify(trader),
        add(order))

name enter(order)
input initial_order
output order
precondition prepared(initial_order)
postcondition entered(order) ∨invalid(order)

name cross(order)
input priced_order
output matched_order
precondition priced(order)
postcondition matched(order) ∨priced(order)

name add(order)
input order
output order_book
precondition uncreated(trade)
postcondition updated(order_book)

name notify(trader)
input trader
output trade_notifcation
precondition created(trade)
postcondition notified(trader)
```

Fig. 8. WPPL specification for new Trade Order Service Requirement

V. RELATED WORKS

Patterns have been attracting more and more attention in the Requirement Engineering (RE) community [10][21][22]. For example, Kamalrudin, Hosking and Grundy [10] reported a technique to improve requirements quality using Essential Use Case (EUC) Interaction Patterns, and developed a CASE tool to support their work. This technique first transformed natural language requirements to semi-formal requirements, i.e., EUC, and extracted abstract interactions from Essential Use Case to derive EUC models. Then EUC models were checked against its matched EUC interaction patterns. The process of EUC technique is similar to our approach. Our approach leverages workflow patterns whereas EUC technique leverages EUC interaction patterns. Sarmiento, Leite and Almentero [22] developed a set of correctness, consistency and completeness (3Cs) patterns for automated scenarios verification. They employ the NFR approach to organize the non-functional requirements related to 3Cs as a catalog of NFRs. While their approach evaluates the properties of scenarios, our approach focuses on the validation of service requirements that have multiple process flows.

The thought that uses NL-oriented tools to elicit requirements has emerged for years, although the automatic extraction of requirements from NL texts is impossible. For example, Gacitua, Sawyer and Gervasi [23] proposed a technique named Relevance driven abstraction identification (RAI) for the identification of key domain abstractions from NL documents, which has been recognized as a useful tool in the analysis of requirements documents. Goldin and Berry [24] developed a prototype NL text abstraction finder named AbstFinder to identify abstractions as well, which is based on traditional signal processing methods. Our approach focuses on capturing the process information from NL texts, rather than the domain abstractions.

VI. DISCUSSIONS AND CONCLUSIONS

Comparing to service design, service requirements does not receive much attention in the development of service-oriented systems. This paper tries to bridge the gap between service requirements analysis and service design, and therefore proposes a systematic approach to model and validate NL-based service requirements documents using workflow patterns.

We demonstrate the applicability of this approach by instantiating it to the evaluation of the Trade Order requirement.

In this paper, we made the following contributions: first, in order to improve the transformation efficiency from NL-based requirements to WPPL, we utilize natural langauge processing techinques to semi-automatically extract the process information from service requirements documents. Second, we apply WPPL to model process-centric service requirements and codify a set of checking rules to each workflow pattern for service requirements validation. Yet, there are still weaknesses in our approach. The extraction process requires that service requirements cannot be documented arbitrarily; rather, they must be documented by following some principles: 1) The description sentences are organized in chronological order; 2) The objects and their manipulations are clear. We will attempt to resolve this problem based on data mining and neural network to improve the precision of NL parsing. Another problem is that our approach only considers the basic workflow patterns The other workflow patterns will be taken into account in the future work. Last but not least, the correctness of the checking rules needs further validation as well.

In conclusion, through presentation, illustration and discussion, this paper has exposed both strengths and weakness of our approach and has shown its applicability in modeling and validating service requirements.

ACKNOWLEDGMENT

This work is sponsored by National Natural Science Foundation of China under Grant No. 61402406, Zhejiang Provincial Natural Science Foundation of China under Grant No. LY13F02010 and No. LY15F02004.

REFERENCES

[1] L. Zhao, J. Wan, P. Jiang, and Y. Qin, "Service Design for Product Lifecycle in Service Oriented Manufacturing," Intelligent Robotics and Applications, vol. 5315, pp 733-742, 2008.

[2] I. J. Jureta, S. Faulkner, and P. Thiran, "Dynamic Requirements Specification for Adaptable and Open Service-Oriented Systems," Proc. 15th IEEE Int. Req. Eng. Conf. (RE'07), IEEE, Oct. 2007, pp. 381-382.

[3] U. Zdun, C. Hentrich, and S. Dustdar, "Modeling process-driven and service-oriented architectures using patterns and pattern primitives," ACM Trans. the Web, ACM, Sept. 2007, pp.1-41.

[4] M. Montali, M. Pesic, W.M.P. Van Der Aalst, F. Chesani, P. Mello, and S. Storari, "Declarative specification and verification of service choreographies," ACM Trans. the Web, vol. 4, no. 1, 2010, pp. 1-62.

[5] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," ACM Trans. on Soft. Eng. & Methodology, vol. 5, no. 3, 1996, pp. 231-261.

[6] X. Li, Z. Liu, and J. He, "Consistency checking of UML requirements," Proc. 10th IEEE Int. Conf. Eng. of Complex Comp. Sys., IEEE, June 2005, pp. 411-420.

[7] R. Bharadwaj, and C. L. Heitmeyer, "Model checking complete requirements specifications using abstraction," Auto. Soft. Eng., vo. 6, no. 1, 1999, pp. 37-68.

[8] V. Gervasi, and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," ACM Trans. Soft. Eng. & Methodology, vol.14, no.3, July 2005, pp. 277-330.

[9] D. Zowghi, and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," Info. & Soft. Tech., vol.45, no.14, Nov. 2003, pp. 993-1009.

[10] M. Kamalrudin, J. Hosking, and J. Grundy, "Improving requirements quality using essential use case Interaction patterns," Proc. Int. Conf. Soft. Eng., IEEE, May 2011, pp. 531-540.

[11] Y. Wang, X. Yang, C. Chang, and A. J. Kavs. "Improving natural language requirements quality using workflow patterns," IEICE Transactions on Information & Systems, vol. 96, no.9, 2013, pp. 2065-2074.

[12] M. Weske, Business Process Management: Concepts, Languages, Architectures, Springer Publishing Company, 2004.

[13] L. Thom, C. Iochpe, and M. Reichert, "Workflow patterns for business process modeling," Proc. Int. Workshop on Business Process Modeling, Springer, Sept. 2007, pp. 349-358.

[14] G. Cai, "Requirement driven service composition: an ontology-based approach," Proc. Int. Conf. IFIP Intelligent Information Processing (IFIP2010), Oct. 2010, pp. 16-25.

[15] S. A. White, "Process modeling notations and workflow patterns," Workflow handbook, 2004, pp. 265-294.

[16] C. Baral, and M. Gelfond, "Reasoning agents in dynamic domains," in Logic-based artificial intelligence, pp. 257-279, Kluwer Academic Publishers Norwell, 2000.

[17] W. Li, "Toward consistency checking of natural language temporal requirements," Proc. Int. Conf. on Automated Software Engineering, Lawrence, KS, USA, 2011, pp. 651-655.

[18] Y. Wang, X. Bai, J. Li, and R. Huang, "Ontology-based test case generation for testing web services," Proc. 8th Int. Symp. Autonomous Decentralized Systems, IEEE, 2007, pp. 43-50.

[19] M. C. De Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," Proc. of 5th Int. Conf. Language Resources and Evaluation, 2006, pp. 449-454.

[20] M. C. De Marneffe, and C. D. Manning, Stanford typed dependencies manual, Technical Report, Stanford University, 2008.

[21] D. Gross, and E. Yu. "From non-functional requirements to design through patterns," Req. Eng. J., vol.6, no.1, Feb. 2001, pp. 18-36.

[22] E. Sarmiento E, J.C.S. do Prado Leite, and E. Almentero, "Using correctness, consistency, and completeness patterns for automated scenarios verification," Proc. IEEE 5th Int. Workshop on Req. Patterns, IEEE, 2015, pp. 47-54.

[23] R. Gacitua R, P. Sawyer P, and V. Gervasi, "On the effectiveness of abstraction identification in requirements engineering," Proc. 18th IEEE Int. Req. Eng. Conf., IEEE, 2010, pp. 5-14.

[24] L. Goldin and D.M. Berry, "AbstFinder: a prototype natural language text abstraction finder for use in requirements elicitation," Automated Software Engineering, vol. 4, no. 4, 1997, pp. 375-412.