

Article

# An Automatic Generation and Verification Method of Software Requirements Specification

Xiaoyang Wei, Zhengdi Wang  and Shuangyuan Yang \*

School of Informatics, Xiamen University, Xiamen 361005, China; konhsan@stu.xmu.edu.cn (X.W.); 24320191152534@stu.xmu.edu.cn (Z.W.)

\* Correspondence: yangshuangyuan@xmu.edu.cn; Tel.: +86-13338459291

**Abstract:** The generation of standardized requirements specification documents plays a crucial role in software processes. However, the manual composition of software requirements specifications is a laborious and time-consuming task, often leading to errors that deviate from the actual requirements. To address this issue, this paper proposes an automated method for generating requirements specifications utilizing a knowledge graph and graphviz. Furthermore, in order to overcome the limitations of the existing automated requirement generation process, such as inadequate emphasis on data information and evaluation, we enhance the traditional U/C matrix by introducing an S/U/C matrix. This novel matrix represents the outcomes of data/function systematic analysis, and verification is facilitated through the design of inspection rules. Experimental results demonstrate that the requirements specifications generated using this method achieve standardization and adherence to regulations, while the devised S/U/C inspection rules facilitate the updating and iteration of the requirements specifications.

**Keywords:** requirements specification; UML diagram; S/U/C matrix; knowledge graph



**Citation:** Wei, X.; Wang, Z.; Yang, S. An Automatic Generation and Verification Method of Software Requirements Specification. *Electronics* **2023**, *12*, 2734. <https://doi.org/10.3390/electronics12122734>

Academic Editor: Claus Pahl

Received: 5 April 2023

Revised: 16 June 2023

Accepted: 16 June 2023

Published: 19 June 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The inception of requirements specification in software engineering can be traced back to the nascent stages of software development, characterized by ad hoc project execution and a lack of standardized or formalized requirements gathering procedures. As the field of software development progressed and the demand for higher quality software intensified, there arose a compelling need for a more structured and formalized approach to requirements gathering and specification.

In the 1970s and 1980s, a number of software development methodologies and frameworks were developed that emphasized the importance of requirements gathering and specification, such as structured analysis and design, information engineering, and entity-relationship modeling. Some more systematic software management methods were also proposed. The waterfall model, known for its linear and sequential nature, places a significant emphasis on upfront requirement gathering and specification. However, this method may encounter challenges in managing evolving requirements or incorporating feedback from stakeholders during later stages of development. V-model emphasizes thorough upfront planning and the validation of requirements through well-defined stages. While this approach provides a structured framework for software development, it can be less flexible when accommodating changing requirements or incorporating feedback during the development process. Agile methodologies, characterized by iterative and incremental development, have gained prominence for their ability to adapt to changing requirements and deliver valuable software in shorter timeframes. However, one of the limitations of agile methods is the potential challenge in maintaining comprehensive and up-to-date requirement documentation throughout the rapid development cycles. This paper aims to overcome these limitations by leveraging automation techniques, providing an efficient and

reliable approach to generate and verify requirements specifications, thereby enhancing the overall software development process.

Software requirement analysis [1,2] serves as the foundational bedrock and primary source for software system design and implementation, with its precision and comprehensiveness constituting a pivotal assurance for software quality [3]. Regrettably, the manual composition of software requirements specifications is laborious and time-intensive, frequently resulting in misconceptions and straying from the actual requirements.

The creation of requirements specifications demands a significant level of skill and expertise from professionals such as systems engineers and UML diagrams can make requirements specifications more intuitive and clearer, thus assisting software development. However, developers may struggle to create precise UML diagrams [4,5], including software architecture and data flow diagrams [6]. Additionally, non-standard requirements specifications can lead to unclear and difficult-to-read requirements documents, causing ambiguity. In addition, small software projects often lack a product manager, which results in a lack of smooth transition from business requirements to functional requirements. Therefore, technicians often develop software projects with subjective understanding, which often leads to rework by customers, resulting in a waste of resources. To avoid these issues and optimize software development work, automatic conversion of business requirements into standard software requirements specifications is necessary.

Indeed, even standardized requirements specification documents are not immune to defects. For instance, a Use/Create (U/C) matrix [7], also referred to as the process/data class matrix in relevant literature, constitutes a tabular representation consisting of a column containing a list of functions and a row containing data classes. The relationship of a function and a data class is represented at the intersection of their corresponding cell, which needs to be filled in. During software requirement modeling, the U/C matrix is a widely accepted approach that links system functions with business data tables. This aids in identifying functional requirements and data specifications for the software's subsequent development and subsystem distribution. However, the U/C matrix exhibits certain limitations in its scope, as it only records the relationship between summarized functional entities and data entities. In practice, the exploration of business requirements during the initial stages of a software project typically entails collaborative efforts involving multiple individuals or even different departments. Consequently, the U/C matrix becomes unsuitable when comparing local requirements across individuals or departments.

Currently, there are three primary approaches for generating standard functional requirements specifications. These include manual UML writing, automatic generation based on structured language and rules, and automatic generation based on knowledge graph. Although manual writing boasts high accuracy, it is time-consuming and often influenced by the developers' subjectivity. The method based on structured language and rules stipulates the strict sentence structure for writing requirements documents, so as to formulate the conversion rules of diagram elements and realize the graphical generation of UML, but it proves challenging for complex structures. Furthermore, the quality of requirements specifications produced through this approach cannot be evaluated quantitatively. The concept of utilizing knowledge graph technology for automated software requirements specifications generation is a novel approach. A knowledge graph consists of a database or a structured data model that captures and represents the relationships between different entities and concepts. The central aspect of UML diagrams is the representation of entities and their connections, which is similar to what knowledge graph represents. Knowledge graph [8,9] technology has achieved considerable progress and efficacy in generating software requirements specifications predominantly in the English language, and there are similar models in the Chinese domain. Nonetheless, there are still some challenges to overcome in this regard:

- I. Existing studies mostly focus on functions, use cases, and classes, with inadequate attention devoted to data-related aspects, including the accuracy of data and the division of functional granularity.

- II. The lack of a comprehensive evaluation concerning the quality of the generated requirements specifications restricts the ability to make necessary adjustments based on the actual circumstances of software development.

This paper presents a proposed resolution to the aforementioned challenges by introducing an automatic generation model for requirements specifications, leveraging the potential of a knowledge graph. The BiLSTM-CRF kg [10] method is used to automatically extract the structured information of functional requirements from the business requirements description corpus and construct a functional structure knowledge graph, while another function-data knowledge graph is built using the structured data in the data tables. Then, merge the two knowledge graphs together. By utilizing the functional entity relationships and data entity relationships within the graph, this model can generate UML diagrams including architecture diagrams and data flow diagrams which can then automatically produce standardized requirements specifications. Additionally, an improved Send/Use/Create (S/U/C) matrix is proposed to address the limitations of the traditional U/C matrix in accommodating coordination and cooperation among multiple departments. This improved matrix can assess the quality of requirements specifications based on the S/U/C situation of data.

## 2. Related Work

A range of methods and models have been introduced in previous research to (semi-) automatically generate requirements specifications.

Miranda, MA et al. [11] proposed the implementation of language of use case to automate models (LUCAM). LUCAM is a specific language that enables the creation of textual use cases and semi-automated generation of use case diagrams, class diagrams, and sequence diagrams using LUCAMTool. However, it only accommodates English business requirements and lacks the assessment of the precision and rationality of UML diagrams. Emeka, B et al. [12] proposed a formal technique for concurrent generation of functional requirements that can help provide a systematic way of accounting for specifying the functional requirements of a software, but it is only limited to the development process of industrial system, and its applicability in other fields has not been verified. Qu, MC et al. [13] introduced a method that can automatically produce standardized requirements and test documentation, addressing problems such as inconsistency, lack of integrity in document-related content, and enhancing efficiency. However, while this method enables tracking of requirements, it does not visually exhibit the results of target labeling. Additionally, the studies mentioned above do not involve creating requirements specifications with regard to data entities.

At the same time, for the testing and evaluation of generating requirements specifications, Mahalakshmi, G et al. [14] emphasized automating the testing process during the software development cycle, starting from the early stages of requirement gathering. Nevertheless, this approach solely considers the use case flow that arises from the use cases and overlooks testing other components in the requirements specification. Tsunoda, T et al. [15] used two specification metrics for SRSs to evaluate their effectiveness to predict future modifications in two actual developments. Franch et al. [16] created a data-based approach, known as Q-Rapids, aimed at enhancing the acquisition, evaluation, and recording of quality requirements in agile software creation. Nonetheless, this approach does not modify and iterate the requirements specification, which fails to facilitate the development team's enhancement in the analysis of requirements and the application of the assessment findings.

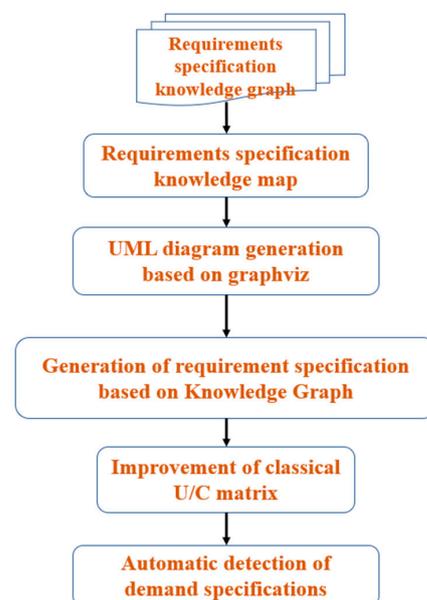
Besides, formal methods are based on strict mathematical foundations, which can generate rigorous, precise, and unambiguous formal constraints, and can be used for model verification and theorem proving, and are complementary to UML. There has been a lot of work on formal methods for writing and verifying requirements. Georgiades et al. [17] provided a novel software tool that attempted to formalize and automate the RE process and extended the use of the tool's SRS document component, which automatically generated well-structured natural language SRS documents. SOFL [18] was proposed to precisely

define functional behavior and incorporate security rules as constraints, which created a solid foundation for implementation and testing. There is also a formal method called RSL-IL [19], allowing for the automated verification and creation of complementary views by performed additional computations on requirements representations, which assist stakeholders in validating requirements. Formal methods generally involve the use of formal languages, mathematical notation, and rigorous techniques. This complexity makes it difficult for non-experts or stakeholders without formal training to understand the requirements generation process and participate effectively in it.

Software development is a knowledge-intensive activity [20]. Some knowledge-based software specification requirements generation methods were also proposed. Avdeenko et al. [21] proposed a hybrid model for classifying requirements using framework ontology and production rules. This method allows property inheritance from parent requirement classes to child classes, and this hierarchy can be used to test the traceability, completeness, and consistency attributes of the requirements specification.

### 3. Structural Information Extraction Model of Functional Requirements

Standard requirements decomposition typically involves breaking down high-level system requirements into smaller and more specific requirements that can be assigned to different components or subsystems of a software system. It helps in organizing and managing complex requirements by dividing them into manageable units. In this paper, software requirements are expressed in the form of knowledge graph. The method first constructs requirements specification knowledge graphs by automatically extracting various entities (manageable units) in the requirements and the relationship between them, and then fuses these graphs into a map. By using the requirements specification knowledge map, it automatically generates UML diagrams, then automatically generates and evaluates the requirements specification. The process is shown in Figure 1:



**Figure 1.** Research route of this paper.

#### 3.1. Requirements Specification Knowledge Graph

The knowledge graph for requirements specification comprises two entities, namely functional entity and data entity. Functional entities have parent–child relationships, while they have creator and user relationships with data entities. Hence, the knowledge graph for requirements specification consists of three triples:

$$\left\{ \begin{array}{l} (\text{functional entity, include, function entity}) \\ (\text{function entity, creator, data entity}) \\ (\text{function entity, user, data entity}) \end{array} \right. \quad (1)$$

In this paper, the BiLSTM-CRF kg method is used to realize the automatic generation of the requirements specification knowledge graph. This method first realizes the embedding of the functional structure knowledge graph from the original business description, and then realizes the embedding of the function-data knowledge graph from the U/C matrix established by business data table description, and finally merges the two graphs to generate a requirements specification map.

In the process of generating the functional structure map, firstly, word segmentation, part-of-speech tagging, functional entity naming tagging, and relationship tagging are performed on the requirement description in the original business description corpus to obtain the functional word segmentation tagging results. Then, entity relationship extraction, entity disambiguation and hidden relationship learning are carried out successively, and the functional structure knowledge graph can be embedded after obtaining the disambiguated functional entity-relationship set and saving it to Neo4j graph database.

In the process of generating the function-data knowledge map, the function-data triplet is obtained by improving the U/C matrix, and then the knowledge graph is automatically embedded by using Neo4j graph database.

### 3.2. UML Diagram Generation Based on Graphviz

After the requirements specification knowledge graph is formed, this paper uses graphviz module to automatically generate architecture diagram according to the parent-child relationship between functional entities and generate data flow diagram according to the relationship between functional entities and data entities.

For a software system, its architecture can be built into a tree model, as shown in Figure 2. Its root node is the whole software system, and the nth layer node corresponds to the N-1 function. Then, for the structural information between functional entities stored in the knowledge graph, breadth first can be adopted, that is, traverse the nodes first, then traverse the relationships, and finally determine the root node.

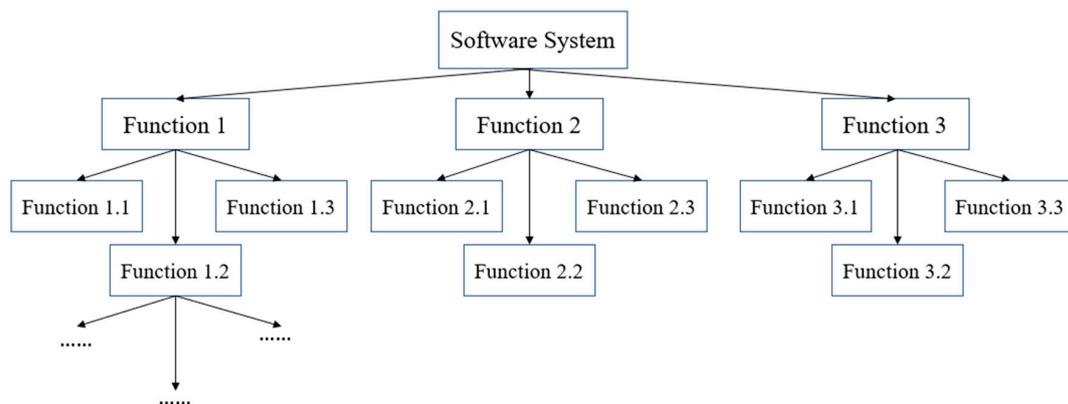


Figure 2. Software system architecture diagram.

In order to mitigate the presence of multiple entities with similar functions in the diagram, the proposed approach employs a two-step process. Initially, during the generation of the software requirements specification map using the BiLSTM-CRF kg method, contextual relationships are leveraged to align functional entities, including pronouns and abbreviations, with the entity-relationship set. This facilitates the acquisition of latent relations between functional entities. Subsequently, in the subsequent phase of generating the UML diagram, a set data structure is employed to store the entities. The functions are then categorized and retained based on the name attribute of the functional entity

within the knowledge graph, leading to the creation of function nodes within the graph. Connections between the nodes are subsequently established in the graph for each pair of triples stored in the graph database, thereby denoting the parent–child relationship between the functions. Lastly, the root node is identified through the detection of the node with a penetration of 0, serving as the foundation for the reconstruction of the entire tree structure.

The whole process is shown in Algorithm 1:

---

**Algorithm 1:** Generation of Software Architecture Diagram

---

**Input:** *KnowledgeGraph*  
**Output:** *Diagram*

```

1: functionSet = KnowledgeGraph.findEntity(label = 'function')
2: triple = KnowledgeGraph.findTriple(head_label = 'function', tail_label = 'function')
3: dot = Digraph(format = 'png')
4: for s in functionSet do
5:     dot.node(name = s['name'], label = s['name'], shape = 'box')
6: end for
7: for t in triple do
8:     dot.edge(t['head']['name'], t['tail']['name'])
9: end for
10: Diagram = dot.view()

```

---

A data flow diagram (DFD) is a graphical tool to describe the data flow in the software system. It marks the logic input and logic output of a system, as well as the processing required to convert the logic input into logic output. In the requirements specification knowledge graph, there are two kinds of relationships between functional entities and data entities: creator and user. Therefore, we can start with the data entity and take the path from its generation to its utilization as the basis for constructing the data flow graph.

To avoid duplication of data entities in the data flow diagram, this study employs a set data structure similar to the software architecture diagram. The first step involves identifying data entities according to their name attribute in the knowledge graph, storing all the data, and generating data nodes in the data flow diagram. The second step is creating functional nodes for each triple pair with head nodes as functional entities and tail nodes as data entities stored in the graph database. Then, the nodes in the graph are connected and marked with the creator or user relationship at the connection.

The whole process is shown in Algorithm 2:

---

**Algorithm 2:** Generation of Data Flow Diagram

---

**Input:** *KnowledgeGraph*  
**Output:** *DFD*

```

1: dataSet = KnowledgeGraph.findEntity(label = 'data')
2: triple2 = KnowledgeGraph.findTriple(head_label = 'function', tail_label = 'data')
3: dot2 = Digraph(format = 'png')
4: for s in dataSet do
5:     dot2.node(name = s['name'], label = s['name'], shape = 'oval')
6: end for
7: for t in triple2 do
8:     dot2.node(name = t['head']['name'], label = t['head']['name'], shape = 'box')
9:     if t['relation']['name'] = 'creator' then
10:         dot2.edge(t['head']['name'], t['tail']['name'], label = 'create')
11:     else
12:         dot2.edge(t['tail']['name'], t['head']['name'], label = 'send')
13:     end if
14: end for
15: DFD = dot2.view()

```

---

### 3.3. Generation of Requirements Specification

According to the relationship between functional entities and between functional entities and data entities in the requirements specification knowledge map, the requirements specification is reconstructed to generate a more standardized requirements specification. At the same time, inserting the generated UML diagram into the document can effectively solve the ambiguity problem of the requirement document.

### 3.4. Improvement of Classical U/C Matrix

The U/C matrix is a crucial tool for analyzing data and functions systematically to divide subsystems. However, the traditional U/C matrix can only be documented in a single format, which does not allow for comparing local requirements across departments and individuals. To address this issue, this study divides the U/C matrix into several Send/Use/Create (S/U/C) matrices. The process involves using a functional entity as a reference point, maintaining only the entire row and data column information labeled with "C", and converting all "U" in the column to "S". Figure 3 shows the primary steps of this process.

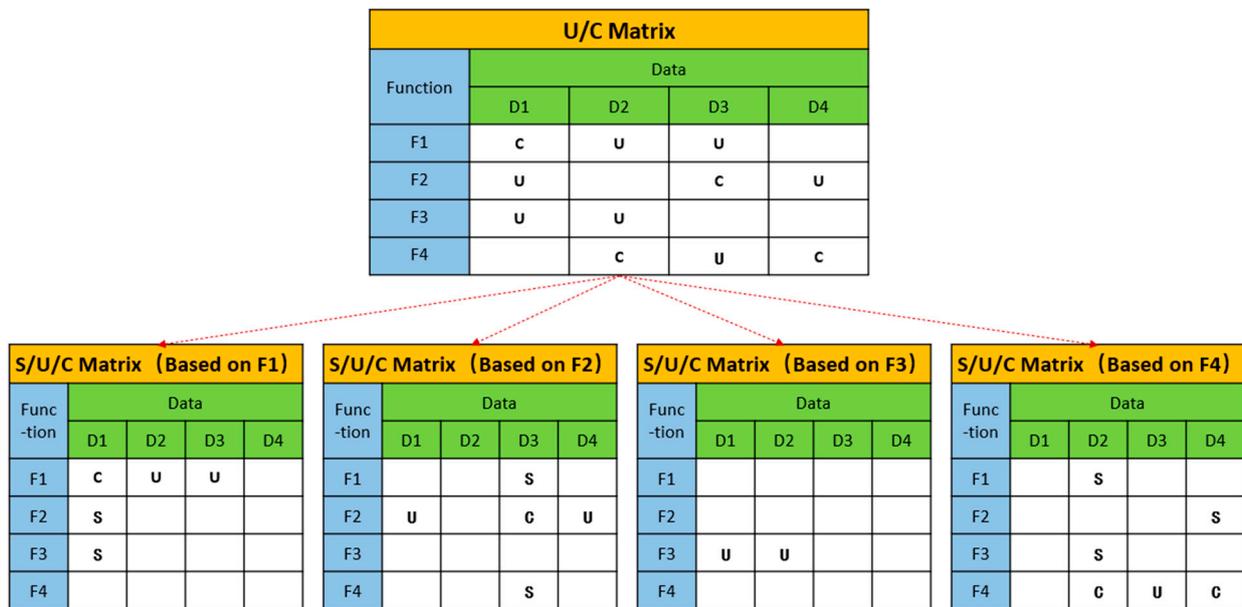


Figure 3. The process of transforming U/C matrix into S/U/C matrix.

By traversing the triples of (function entity, relationship, data entity) in the knowledge graph, this paper generates a U/C matrix with data as rows and function as columns, and converts it into S/U/C matrix according to the above operations.

The whole process is shown in Algorithm 3. Specifically, the steps to optimize the U/C matrix to S/U/C matrix are as follows: for each functional entity, a matrix with the same specifications as the U/C matrix is established separately. For the data generated by this function, the corresponding position under this function dimension in the S/U/C matrix is recorded as "C"; for the data used by this function, the corresponding position under the functional dimension in the S/U/C matrix is recorded as "U" (if the corresponding position is already "C", then "CU" is recorded at the corresponding position); for the data generated by this function and used by other functions, the corresponding position under this function dimension in the S/U/C matrix is recorded as "S", thus forming a three-dimensional S/U/C matrix.

**Algorithm 3:** Improvement of classical U/C matrix**Input:** *KnowledgeGraph***Output:** *SUC*

```

1: //Create U/C Matrix
2: triple3 = KnowledgeGraph.findTriple(head_label = 'function', tail_label = 'data')
3: functionSet = set(triple3['head'])
4: dataSet = set(triple3['tail'])
5: for t in triple3 do
6:   if t['relation']['name'] == 'creator' then
7:     if UC[t['head']['name']][t['tail']['name']] == 'U' then
8:       UC[t['head']['name']][t['tail']['name']] = 'CU'
9:     else
10:      UC[t['head']['name']][t['tail']['name']] = 'C'
11:    end if
12:  else
13:    if UC[t['head']['name']][t['tail']['name']] == 'C' then
14:      UC[t['head']['name']][t['tail']['name']] = 'CU'
15:    else
16:      UC[t['head']['name']][t['tail']['name']] = 'U'
17:    end if
18:  end if
19: end for
20: //ChangeU/CMatrixtoS/U/CMatrix
21: for f in functionSet do
22:   Cset = set()
23:   for i in functionSet do
24:     for j in dataSet do
25:       if i == f then
26:         UCS[f][i][j] = UC[i][j]
27:         if UC[i][j] == 'C' or UC[i][j] == 'CU' then
28:           Cset.add(j)
29:         end if
30:       end if
31:     end for
32:   end for
33:   for k in Cset do
34:     for p in functionset do
35:       if UC[p][k] == 'U' then
36:         UCS[f][p][k] = 'S'
37:       end if
38:     end for
39:   end for
40: end for

```

**3.5. Automatic Detection of Requirements Specifications**

After the generation of S/U/C matrix, this paper generates the inspection principle of S/U/C matrix according to the inspection principle of U/C matrix, so as to test the correctness, completeness and consistency of requirements specifications.

The correctness is checked with the following four rules:

- (1) Sole produce rule of data entity (CTR-SPR): for any data table entity in the requirements specification map, it has and only has one Creator. Its mathematical definition is:

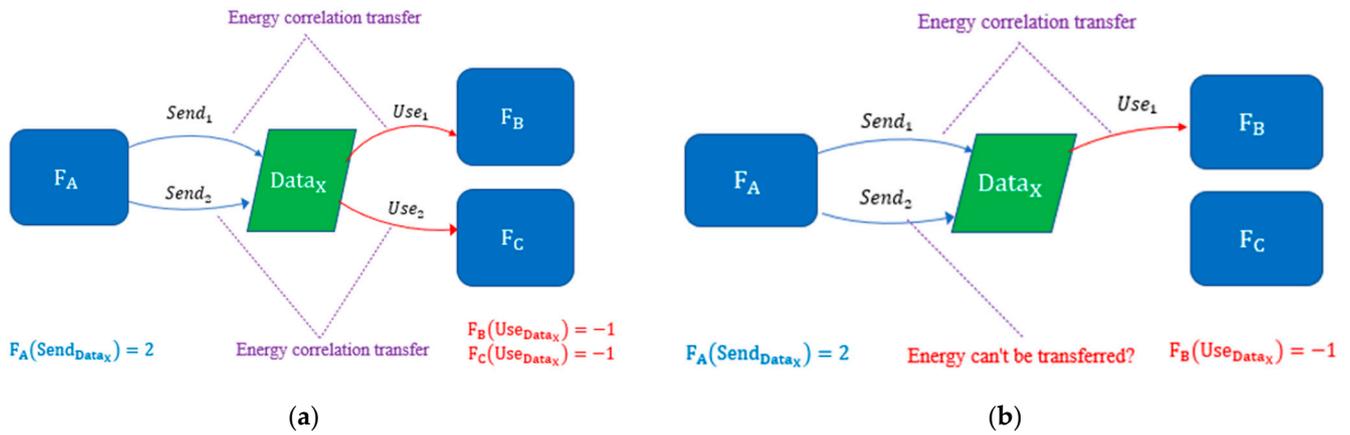
$$F_D(\text{Creator}) = 1 \quad (2)$$

$F_D$  represents the processing of data entities.

- (2) Multi-user rule of data entity (CTR-MUR): for any data table entity in the requirements specification graph, it has at least one user. Its mathematical definition is:

$$F_D(Use) \geq 1 \tag{3}$$

- (3) Energy conservation rule of data entity (CTR-ECR): the schematic diagram of energy conservation of data entity is shown in Figure 4.



**Figure 4.** Data entity energy transfer relationship. (a) Correct data entity energy transfer relationship; (b) Wrong data entity energy transfer relationship.

In the process of demand research, many people often conduct research synchronously or need to investigate multiple business departments. Each researcher or each business department has different opinions on whether a specific data table is generated or used, and there is often negligence. This will result in the omission of functional requirements or the lack of data flow. From the perspective of data in the knowledge map, the generation and use of a certain type of data are corresponding. That is, for any data table entity, the number of the received results corresponds to the number of the sent results. If not, then something must have gone wrong in the data flow.

In order to solve this problem, the project defines that the energy emitted (Number of  $Send_{data}$ ) by any data table entity in the requirements specification map is opposite to the energy used (Number of  $Use_{data}$ ), and the quantity is equal. Its mathematical definition is:

$$\sum_{\substack{1 \leq i \leq N \\ 1 \leq j \leq M}} (F_i(Send_{data_i}) + F_j(Use_{data_j})) = 0 \tag{4}$$

- (4) Energy keeping rule of functional entity (CTR-EKR): for any functional entity in the requirements specification graph, the sum of the absolute value of energy sent and received is greater than or equal to 1. If it is 0, it indicates that the function may have missing data transmission or be a single independent function (which is relatively rare). Its mathematical definition is:

$$|F_F(Send_{data})| + |F_D(Use_{data})| \geq 1 \tag{5}$$

$F_F$  represents the processing of data entities. Similarly, it is easy to draw the following conclusions:

$$\begin{cases} \text{if } (|F_F(Send_{data})| + |F_D(Use_{data})| = 0), \text{ maybe missing data transfer} \\ \text{if } (|F_F(Send_{data})| \geq 1 \cap |F_D(Use_{data})| = 0), \text{ is basic data module} \\ \text{if } (|F_F(Send_{data})| \geq 1 \cap |F_D(Use_{data})| \geq 1), \text{ is business processing module} \\ \text{if } (|F_F(Send_{data})| = 0 \cap |F_D(Use_{data})| \geq 1), \text{ is statistic analysis module} \end{cases} \tag{6}$$

Correctness Perfect Rules (CPR) are used to check completeness of requirements specifications. The completeness test mainly includes three parts: function inspection test, data table inspection test and parent–child functional relationship inspection test. That is, the function test list *FuncTestCaseList*, the data test list *DataTestCaseList* and the parent–child functional relationship test list *PFuncTestCaseList* are added automatically and manually. These test cases are automatically queried in the generated requirements specification map to find out whether there are any omissions. Its mathematical definition is:

$$\begin{cases} \text{if}(F_{exist}(\exists func \in RKG, \forall func \in \text{FuncTestCaseList})) = true \\ \text{if}(F_{exist}(\exists data \in RKG, \forall data \in \text{DataTestCaseList})) = true \\ \text{if}(F_{exist}(\exists pfunc \in RKG, \forall pfunc \in \text{PFuncTestCaseList})) = true \end{cases} \quad (7)$$

*RKG* represents the requirements specification knowledge graph.  $F_{exist}$  indicates whether the search result in *RKG* exists, if it exists, the test is determined to be successful, otherwise it is not successful and the completeness is lacking.

Consistency Test (CST) mainly includes two tests: the path attack test (main process test) from function A to function B and the automatic generation function deployment test of DFD. That is to determine whether the main process extracted by *RKG* is completely consistent with the core main process of the system, and whether the DFDs generated by *RKG* extraction at all levels are consistent with the real business situation. By examining the judgment result, it can be determined if there are any discrepancies in *RKG*.

## 4. Experimental Results and Analysis

### 4.1. Dataset Source and Experimental Environment

The data used in this paper were mainly the system requirement documents in the actual software development project collected online and offline. A total of 150 system requirements documents were obtained and used to train the BiLSTM-CRF kg model and generate software requirements specification maps. Among these maps, 15 were randomly chosen for quality inspection and software requirements specification generation.

This experiment used the *graphviz* module based on Python to generate UML diagrams, so as to realize the generation of requirements documents, define the requirements specification inspection principle and realize the improvement of U/C matrix and the inspection of requirements based on data flow. The specific software and hardware configuration of the experimental environment is shown in Table 1.

**Table 1.** Configuration of experimental environment.

Environment Type	Details	
Hardware Environment	CPU	Intel(R) Xeon(R) CPU E5-2620 v4 @2.10 GHz × 32
	GPU	GeForce GTX 1080 Ti/PCIe/SSE2
	Memory	64 G
	Video Memory	10.91 G
Software Environment	Operating System	Ubuntu20.04
	Tool Kit	<i>graphviz</i>
	Development Language	Python 3.6
	Development Tool	JetBrains PyCharm Community Edition 2020.2

### 4.2. Typical Experimental Results and Evaluation

Firstly, the experiment needs to generate standard requirements specification through the knowledge graph stored in the graph database Neo4j. The knowledge graph is shown in Figure 5.

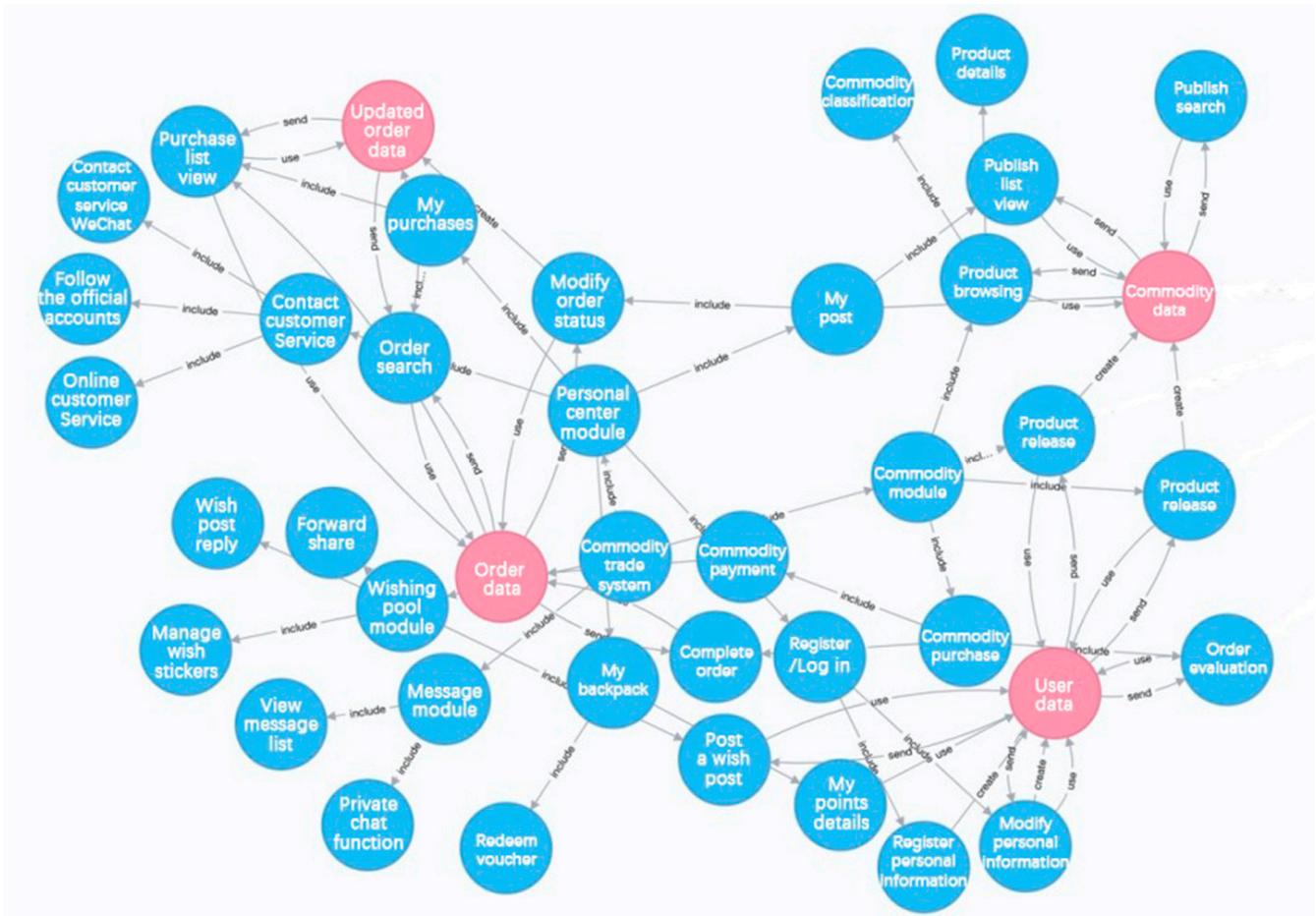


Figure 5. Knowledge Graph.

The graph includes both function and data entities. There is a parent–child relationship between function entities and a create and use relationship between function entities and data entities. The experiment generates the architecture diagram of the software system according to the parent–child relationship between functional entities, and generates the data flow diagram according to the create/use relationship between functional entities and data entities. Taking the “second-hand trading platform” as an example, the architecture diagram generated through the knowledge graph is shown in Figure 6 and the data flow diagram is shown in Figure 7a.

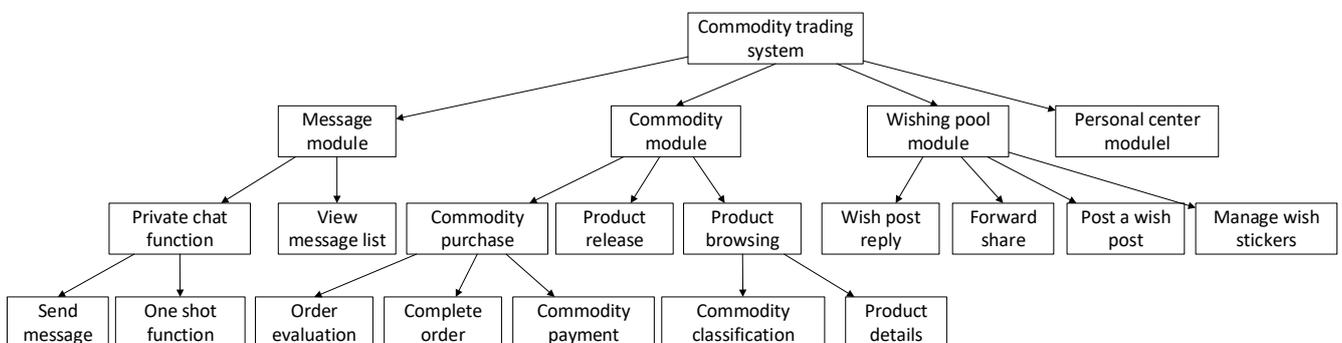
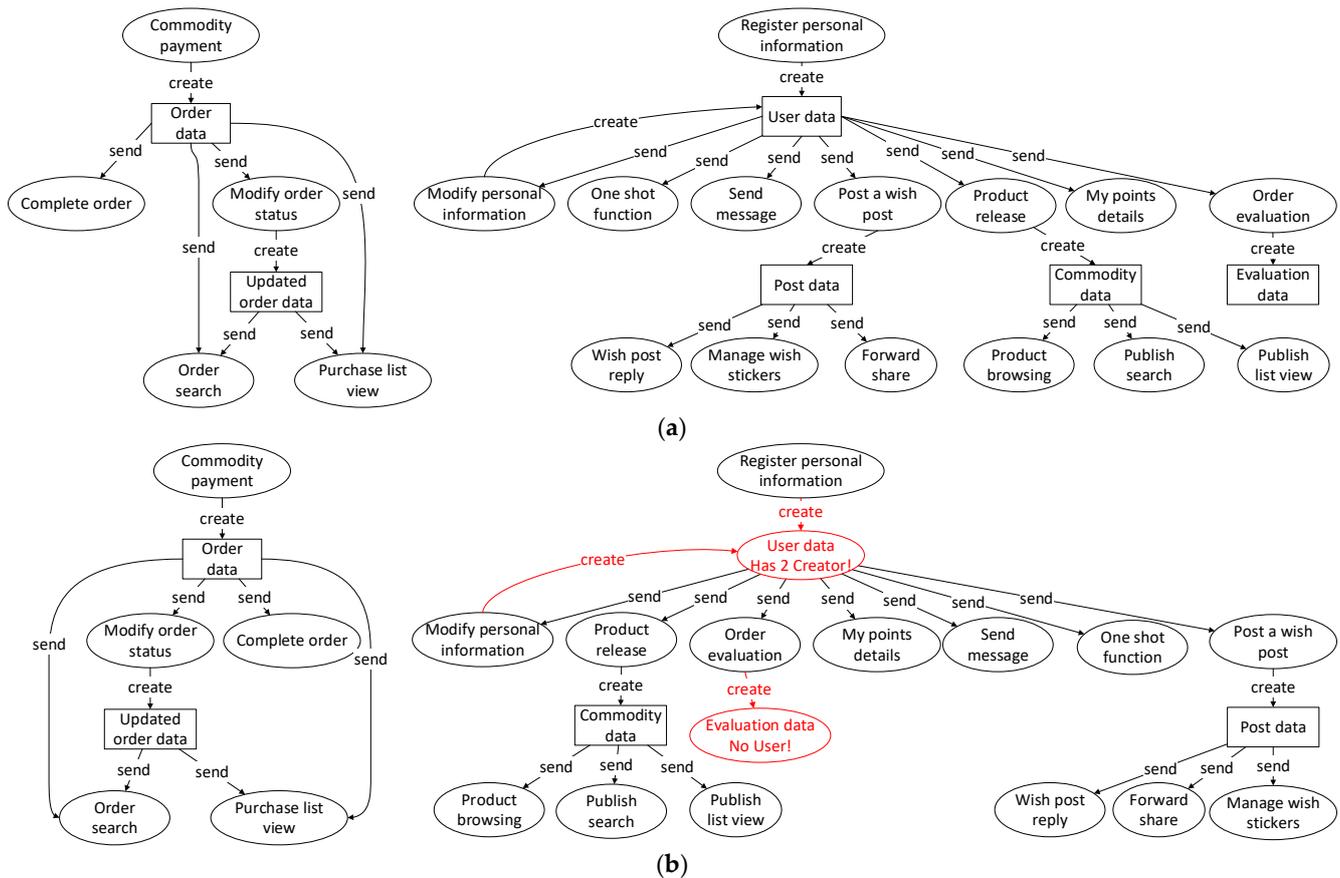


Figure 6. Software architecture diagram.



**Figure 7.** Data Flow Diagrams. (a) Generated data flow diagram; (b) Inspection result of requirements specifications. Places marked in red show possible errors in DFDs.

By drawing UML diagrams and describing the relationship between various functions and data, standard requirements specification documents can be generated.

At the same time, the experiment generates U/C matrix through the relationship between functional entities and data entities, and then decomposes it into S/U/C matrix. The U/C matrix and the decomposed S/U/C matrix are shown in Figure 8.

After the S/U/C matrix is generated, the experiment tests the S/U/C matrix according to Formulas (2)–(5). First of all, the experiment needs to automatically generate test cases according to these rules, using the error presumption method in the software testing method.

As for correctness tests, the experiment extracts a list of data tables from the S/U/C matrix as the source of test cases for checking energy relationships in the CTR-SPR, CTR-MUR, and CTR-ECR rules. It tests whether the data table only has one creator, whether there are users, and whether the emitted energy is equal to the received energy. Additionally, a list of functions is extracted from the business requirement description as the source of test cases for checking energy relationships in the CTR-EKR rule, focusing on the principle of energy keeping. The results of correctness tests are shown in Figure 9. For CTR-SPR, CTR-MUR, and CTR-ECR rules, the number of test cases is the number of data tables. For the CTR-EKR rule, the number of test cases is the number of functional entities. It shows that the “use” of the evaluation information node is 0, that is, the CTR-MUR rule is not satisfied. This shows that during the demand research process, the demand analyst may have missed a certain functional module; at the same time, the classification information does not satisfy the principle of energy conservation, which also shows that there may be a lack of function during the demand research process.

(a) U/C Matrix

	Commodity data	Order data	Updated order data	User data	Post data	Evaluation data
Product release	C			U		
Product browsing	U					
Publish list view	U					
Publish search	U					
Commodity payment		C				
Complete order		U				
Purchase list view		U	U			
Order search		U	U			
Modify order status		U	C			
Register personal information				C		
Order evaluation				U		C
Post a wish post				U	C	
Send message				U		
One shot function				U		
Modify personal information				CU		
My points details				U		
Wish post reply					U	
Forward share					U	
Manage wish stickers					U	

	Commodity data	Order data	Updated order data	User data	Post data	Evaluation data
Product release						
Product browsing						
Publish list view						
Publish search						
Commodity payment						
Complete order						
Purchase list view						
Order search						
Modify order status						
Register personal information						
Order evaluation						S
Post a wish post						S
Send message						S
One shot function						S
Modify personal information						CU
My points details						S
Wish post reply						S
Forward share						S
Manage wish stickers						S

	Commodity data	Order data	Updated order data	User data	Post data	Evaluation data
Product release	C			U		
Product browsing	S					
Publish list view	S					
Publish search	S					
Commodity payment						
Complete order						
Purchase list view						
Order search						
Modify order status						
Register personal information						
Order evaluation						S
Post a wish post						S
Send message						S
One shot function						S
Modify personal information						CU
My points details						S
Wish post reply						S
Forward share						S
Manage wish stickers						S

	Commodity data	Order data	Updated order data	User data	Post data	Evaluation data
Product release						
Product browsing						
Publish list view						
Publish search						
Commodity payment						
Complete order						
Purchase list view						
Order search						
Modify order status						
Register personal information						
Order evaluation						S
Post a wish post						S
Send message						S
One shot function						S
Modify personal information						CU
My points details						S
Wish post reply						S
Forward share						S
Manage wish stickers						S

(b) S/U/C Matrix

Figure 8. U/C Matrix to S/U/C Matrix.

Use case data	Expected value	Actual value
User data	1	2
Order data	1	1
Updated order data	1	1
Post data	1	1
Evaluation data	1	1
Commodity data	1	1

(a)

Use case data	Expected value	Actual value
User data	>=1	7
Order data	>=1	3
Updated order data	>=1	2
Post data	>=1	3
Evaluation data	>=1	0
Commodity data	>=1	4

(b)

Use case data	Energy emitted	Energy used
User data	7	6
Order data	3	3
Updated order data	2	2
Post data	3	3
Evaluation data	0	0
Commodity data	4	4

(c)

Use case data	Energy emitted	Energy used
Product release	1	1
Commodity payment	0	0
Modify order status	1	1
Post a wish post	1	1
Order evaluation	1	1
Modify personal information	1	1
Register personal information	0	0
One shot function	0	1
My points details	0	1
...		

(d)

Figure 9. Results of correctness tests. Red marks indicate that the test results are contrary to inspection rules. (a) Results of CTR-SPR inspection; (b) Results of CTR-MUR inspection; (c) Result of CTR-ECR inspection; (d) Results of CTR-EKR inspection.

When employing completeness rules for testing purposes, three distinct types of test cases become necessary: functional module test cases, data table test cases, and functional parent-child relationship test cases. The functional module test cases are derived from an automatically generated list of business functions extracted from the business requirements description. Their primary purpose is to assess the presence or absence of any missing system function modules. The data table test cases, on the other hand, stem from the U/C

matrix and are employed to determine the existence or absence of data tables. Lastly, the functional parent–child relationship test cases are derived from the parent–child relationship list found within the automatically generated business function list from the business requirements description. These test cases are instrumental in evaluating the presence or absence of parent–child relationships between system function modules. The results of completeness tests are shown in Figure 10.

Use case data	Expected result	Actual result
Commodity data	TRUE	TRUE
Order data	TRUE	TRUE
Updated order data	TRUE	TRUE
User data	TRUE	TRUE
Evaluation data	TRUE	TRUE
Post data	TRUE	TRUE
Registration information	FALSE	FALSE
Comment data	FALSE	FALSE

(a)

Use case data	Expected result	Actual result
Product release	TRUE	TRUE
It	FALSE	FALSE
This function	FALSE	FALSE
Product browsing	TRUE	TRUE
Contact consumer service	TRUE	TRUE
Telephone customer service	FALSE	FALSE
...	...	...

(b)

Use case data	Expected result	Actual result
{father: 'Commodity module', son: 'Product release'}	TRUE	TRUE
{father: 'Wishing pool module', son: 'Manage wish stickers'}	TRUE	TRUE
{father: 'Commodity module', son: 'Publish search'}	TRUE	FALSE
{father: 'Message module', son: 'View message list'}	TRUE	TRUE
{father: 'Contact consumer service', son: 'Follow the official accounts'}	TRUE	TRUE
...	...	...

(c)

**Figure 10.** Results of completeness tests. Places marked in red indicate that the actual results did not meet expectations. (a) Results of data table case test; (b) Results of functional module case test; (c) Results of functional parent–child relationship case test.

Observing Figure 10, it is evident that there is a wrong inclusion relationship (error\_include) between the publish search and commodity module nodes. Upon examining the original business description corpus, it is clear that there is a parent–child relationship between these modules, but it is not reflected in the software requirements specification map. This highlights the possibility of errors in the construction of the software requirements specification map for the commodity and publishes search modules.

The consistency test includes a functional path test and a DFD inclusion test, which, respectively require some correct business process paths and correct DFD diagrams as test cases, and then verify whether they are consistent in the requirements specification map. The results of the consistency test are shown in Figure 11.

The use case data of the function path use case is an array, and the order of the array of elements represents the order of the test function path; while the use case data of the DFD containing use cases is a JSON structure, its data attribute represents the data table, and the route attribute is an array, indicating the direction of the data flow. It can be seen from Figure 11 that there is a lack of data transfer relationship between the release of wishing stickers and the release of products. The reason for this kind of problem may be an error in the process of requirement research or an error in the process

of embedding the requirements specification map. Through the inspection of the original S/U/C matrix, it was found that the problem of missing data flow direction existed in the demand research process.

Use case data	Expected result	Actual result
{data: 'Post data', route: ['Post a wish post', 'Wish post reply']}	TRUE	TRUE
{data: 'Order data', route: ['Commodity payment', 'Order search']}	TRUE	TRUE
{data: 'User data', route: ['Modify personal information', 'Send message']}	TRUE	TRUE
{data: 'User data', route: ['Post a wish post', 'Product release']}	TRUE	FALSE
...	...	...

(a)

Use case data	Expected result	Actual result
['Commodity module', 'Commodity purchase', 'Complete order']	TRUE	TRUE
['Personal center module', 'Register personal information']	TRUE	TRUE
['Commodity trade system', 'Order search']	TRUE	TRUE
['Commodity trade system', 'Personal center module', 'My purchases', 'Order search']	TRUE	TRUE
...	...	...

(b)

Figure 11. Results of Consistency test. Places marked in red indicate that the actual results did not meet expectations. (a) Functional path test results; (b) DFD inclusion test results.

In addition, the method in this paper can display the test results in a DFD, as shown in Figure 7b. Based on the results of the automated validation analysis, the requirements specification can be revised and the version can then be updated.

Finally, this paper conducts statistical analysis on the quality inspection results of the generated requirements specification maps of 15 software systems, as shown in Table 2 and Figure 12.

Table 2. Quality inspection use case test results.

Use case Type	The Total Number of Use Cases of This Type	Number of Use Cases Passed	Passing Rate
Correctness test use case	628	599	95.38%
Completeness test use case	1563	1524	97.50%
Consistency test use case	287	273	95.12%

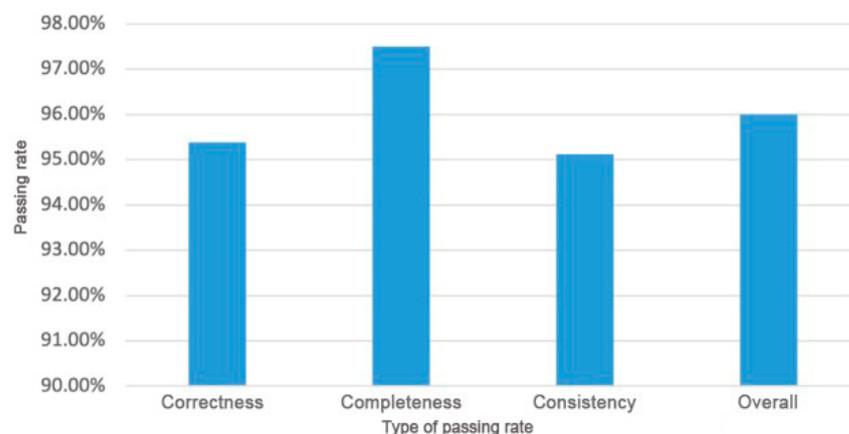


Figure 12. Requirements specification map quality inspection use case test passing rate.

### 5. Conclusions

The purpose of this paper is to address the challenges associated with inconsistencies in business descriptions, low writing efficiency, and error-prone processes during manual

software requirements specification compilation. To do so, we utilize the BiLSTM-CRF kg model to automatically create a software requirements specification map and generate the corresponding software requirements specification. Additionally, we enhance the U/C matrix used in system analysis by introducing the S/U/C matrix, which facilitates cross-departmental and individual comparisons of local demand survey results. To ensure the quality of the software requirements specification map, a quality inspection method is proposed that leverages the energy transfer relationships between entities in the map. This inspection method includes a series of test cases focusing on correctness, completeness, and consistency. By quickly locating and labeling errors on the DFD, this method allows for the timely resolution of issues that arise during the software requirement research and specification preparation process. The study randomly chose 15 out of 150 software requirements specification maps generated by the BiLSTM-CRF kg model for quality evaluation. The results indicate that the use case test of the requirements specification map has a passing rate of 96%, and the method presented in the paper is effective in generating and updating software requirements specifications.

Although this paper realizes the automatic generation and quality inspection of the software requirements specification, the quality of the software requirements specification map has not been fully evaluated by classification and quantification, and the severity of different errors cannot be distinguished, which requires further study. Additionally, the timing issue during the data flow process will be further considered in future work.

**Author Contributions:** Conceptualization, S.Y., Z.W. and X.W.; methodology, X.W., Z.W. and S.Y.; software, X.W. and Z.W.; validation, Z.W. and X.W.; formal analysis, Z.W.; investigation, X.W.; resources, S.Y.; data curation, X.W.; writing—original draft preparation, X.W.; writing—review and editing, Z.W.; visualization, X.W.; supervision, S.Y.; project administration, S.Y.; funding acquisition, S.Y. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by the Natural Science Foundation of Fujian Province of China (No. 2022J01003).

**Data Availability Statement:** The data presented in this study are available on request from the corresponding author. The data are not publicly available due to further research plans.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Hung, B.; Omori, T.; Ohnishi, A. Ripple effect analysis of data flow requirements. In Proceedings of the 14th International Conference on Software Technologies, Prague, Czech Republic, 26–28 July 2019; pp. 262–269.
2. Boyarchuk, A.; Pavlova, O.; Bodnar, M.; Lopatto, I. Approach to the Analysis of Software Requirements Specification on Its Structure Correctness. In Proceedings of the IntelITSIS, Khmelnytskyi, Ukraine, 10–12 June 2020; pp. 85–95.
3. Liu, C.; Zhao, Z.; Zhang, L.; Li, Z. Automated conditional statements checking for complete natural language requirements specification. *Appl. Sci.* **2021**, *11*, 7892. [[CrossRef](#)]
4. Braude, E.; Van Schooneveld, J. Incremental UML for agile development with PREXEL. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, Gothenburg, Sweden, 27 May–3 June 2018; pp. 310–312.
5. Torre, D.; Labiche, Y.; Genero, M.; Baldassarre, M.T.; Elaasar, M. UML diagram synthesis techniques: A systematic mapping study. In Proceedings of the 10th International Workshop on Modelling in Software Engineering, Gothenburg, Sweden, 27–28 May 2018; pp. 33–40.
6. Heayyoung, J.; Omori, T.; Ohnishi, A. Ripple effect analysis method of data flow diagrams in modifying data flow requirements. In Proceedings of the 10th Knowledge-Based Software Engineering: 2018: Proceedings of the 12th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2018), Corfu, Greece, 27–30 August 2018; pp. 1–11.
7. International Business Machines Corporation. *Data Processing Division. Business Systems Planning: Information Systems Planning Guide*; IBM: Armonk, NY, USA, 1978.
8. Lin, J.; Zhao, Y.; Huang, W.; Liu, C.; Pu, H. Domain knowledge graph-based research progress of knowledge representation. *Neural Comput. Appl.* **2021**, *33*, 681–690. [[CrossRef](#)]
9. Kejriwal, M.; Sequeda, J.F.; Lopez, V. Knowledge graphs: Construction, management and querying. *Semant. Web* **2019**, *10*, 961–962. [[CrossRef](#)]
10. Wang, Z.; Pan, J.-S.; Chen, Q.; Yang, S. BiLSTM-CRF-KG: A Construction Method of Software Requirements Specification Graph. *Appl. Sci.* **2022**, *12*, 6016. [[CrossRef](#)]

11. Miranda, M.A.; Ribeiro, M.G.; Marques-Neto, H.T.; Song, M.A.J. Domain-specific language for automatic generation of UML models. *IET Softw.* **2018**, *12*, 129–135. [[CrossRef](#)]
12. Emeka, B.; Liu, S. A Formal Technique for Concurrent Generation of Software’s Functional and Security Requirements in SOFL Specifications. In *International Workshop on Structured Object-Oriented Formal Language and Method 2019*; Springer International Publishing: Cham, Switzerland, 2020; pp. 13–28.
13. Qu, M.; Wu, X.; Tao, Y.; Liu, Y. Research on generating method of embedded software test document based on dynamic model. *IOP Conf. Ser. Mater. Sci. Eng.* **2018**, *322*, 062018. [[CrossRef](#)]
14. Mahalakshmi, G.; Vijayan, V.; Antony, B. Named entity recognition for automated test case generation. *Int. Arab J. Inf. Technol.* **2018**, *15*, 112–120.
15. Tsunoda, T.; Washizaki, H.; Fukazawa, Y.; Inoue, S.; Hanai, Y.; Kanazawa, M. Empirical study on specification metrics to predict volatility and software defects. In *Proceedings of the TENCON 2018-2018 IEEE Region 10 Conference*, Jeju, Republic of Korea, 28–31 October 2018; pp. 2479–2484.
16. Franch, X.; Gómez, C.; Jedlitschka, A.; López, L.; Martínez-Fernández, S.; Oriol, M.; Partanen, J. Data-driven elicitation, assessment and documentation of quality requirements in agile software development. In *Proceedings of the Advanced Information Systems Engineering: 30th International Conference, CAiSE 2018, Tallinn, Estonia, 11–15 June 2018*; pp. 587–602.
17. Georgiades, M.G.; Andreou, A.S. Automatic generation of a software requirements specification (SRS) document. In *Proceedings of the 2010 10th International Conference on Intelligent Systems Design and Applications, Cairo, Egypt, 29 November–1 December 2010*; pp. 1095–1100.
18. Emeka, B.; Liu, S. A Formal Technique for Concurrent Generation of Software’s Functional and Security Requirements in SOFL Specifications. In *International Workshop on Structured Object-Oriented Formal Language and Method*; Springer International Publishing: Cham, Switzerland, 2019; pp. 13–28.
19. de Almeida Ferreira, D.; Da Silva, A.R. Formally specifying requirements with RSL-IL. In *Proceedings of the 2012 Eighth International Conference on the Quality of Information and Communications Technology, Lisbon, Portugal, 3–6 September 2012*; pp. 217–220.
20. van Vliet, H. Knowledge sharing in software development. In *Proceedings of the 10th International Conference on Quality Software (QSIC), Zhangjiajie, China, 14–15 July 2010*; p. 2.
21. Avdeenko, T.V.; Pustovalova, N.V. The ontology-based approach to support the requirements engineering process. In *Proceedings of the 2016 13th International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE), Novosibirsk, Russia, 3–6 October 2016*; pp. 513–518.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.