Formal Modeling of Airborne Software High-Level Requirements Based on Knowledge Graph

Wenjuan Wu, Dianfu Ma, Yongwang Zhao, and Xianqi Zhao

School of Computer Science and Engineering,
Beijing University of Aeronautics and Astronautics, Beijing, 100191
{wuwj,dfma,zhaoyw,zhaoxq}@act.buaa.edu.cn

Abstract. Airborne airworthiness certification DO-178C software release proposes a higher safety and reliability demands of airborne software. This raises great challenges to airborne software modeling and verification. In order to achieve airborne software high-level requirements objectives, we propose a formal method of modeling high-level requirements based on knowledge graph. The method gives a formal language to describe knowledge graph and constructs knowledge graph collaboratively. Then we represents high-level requirements by causal model and formal modeling of high-level functional requirements and non-functional requirements by knowledge graph. These improve the requirement traceability, namely these are helpful to trace the high-level requirements to system requirements so as to achieve high-level requirements' traceability objective that DO-178C demands. Additionally, we provide the modeling tool for domain experts to construct knowledge graph collaboratively and realize their high-level requirements modeling. We also give some high-level requirements verification. These are significant to generate safe, reliable, accurate and high-quality airborne software.

Keywords: Formal modeling, High-level requirements, Airborne software, Knowledge graph, DO-178C.

1 Introduction

With sharply increasing software scale and complexity, development of avionics software faces huge challenges including rising safety requirement, increasing verification cost and shortening time to market demands [1]. In order to ensure airborne software safety, it is necessary to provide adequate safety certification for software before put into use. The verification criteria currently used are aviation airworthiness certification standards DO-178B [2] and DO-178C [3] which are made by The Radio Technical Commission for Aeronautics, RTCA. DO-178C [3] proposes that system requirements, hardware interfaces, system architecture from system life cycle process and software development plan, software requirement standard from software plan process are inputs of the model. When

R. Buchmann et al. (Eds.): KSEM 2014, LNAI 8793, pp. 258–269, 2014.

conversion rules are determined, these inputs will be used to develop high-level requirements. The output of the process is software requirement data.

Current main modeling languages of airborne software are AADL [8], UML, UML MARTER [9] and so on. The main requirement modeling approach is use case diagram and use case description of UML. However, UML has some drawbacks [4]. First, UML lacks of process guidance. It is not a method but a modeling language and does not define process guidance. Then we cannot develop a really good system and guarantee the quality of the software only by UML. Second, UML is too complicated. Third, there is not an effective and rigorous method to verify and test software system modeled by UML. At the same time, lack of accuracy will reduce software quality. Also, requirement documents are tedious in traditional and high-level requirements traceability is hard to achieve. Thus leads workload and can't guarantee accurate traceability to meet the need of high quality software.

We apply knowledge graph approach to high-level requirements modeling of airborne software. The inputs of system requirements are perception concepts and the outputs of system requirements are actuation concepts. How to get actuation concepts from perception concepts is a black box problem. Knowledge graph turn the black box problem into white box problem and get high-level requirements. Traceability can be improved and formal modeling of high-level requirements is helpful for us to verify our high-level requirements.

This paper makes the following contributions:

a. It proposes a new formal method of high-level requirements modeling of airborne software called RMKG(Requirement Modeling based on Knowledge Graph).

b. It displays requirements traceability which is beneficial to trace the system requirements from high-level requirements .

c. Its visual knowledge graph modeling is intuitive and helpful for domain experts to communicate when constructing knowledge graph.

The rest of this paper is organized as follows. Some related work is given in Section 2. We give detailed description of knowledge graph in section 3. Section 4 describes the high-level requirements modeling and verification. Section 5 contains experimental results to evaluate our modeling tool. We draw conclusions and propose our future work in section 6.

2 Related Work

Knowledge graph is used to represent an idea, event, situation or circumstance described by a trend graph, which consists of nodes to represent concepts and links to represent the conceptual relationships which is an instrument that represents some knowledge as a way to represent the logical structure of knowledge described in natural language [5]. Nguyen-Vu Hoang proposes a representation of the knowledge on relationships existing between symbolic objects in a collection of images. They present a graph based representation of this knowledge and its associated operations and properties [6]. While we apply knowledge graph and

knowledge inference to our requirements modeling. Harry S. Delugach apply conceptual graphs to acquiring software requirements. They use conceptual graphs to represent requirements knowledge, repertory grids to acquire requirements knowledge and formal concept analysis to form requirements concept [7]. But they left some problems such as requirement traceability which we'll improve in this paper.

3 Knowledge Graph

3.1 Formal Description of Knowledge Graph

Definition 1: *Causal model* summarizes the interaction between environment and computer system, identifies the relationship between the initial concept and other concepts based on the first-order logic and collections to get the architecture of a system.

Definition 2: *Causality* is defined as $< x_i, x_{i+1}, ..., x_j, y_i >$. Output y_i is calculated by internal behavior of the system based on the inputs $x_i, x_{i+1}, ..., x_j$. Such even forms a causality.

Definition 3: In airborne system, output data must be calculated by input data and intermediate data got by input data. Therefore, there exists a *causal chain* like $< x_i, x_{i+1}, ..., x_m, r >< x_{m+1}, ..., x_j, r, s >< s, t >< t, y >$ which can describe how to get output data from input data and intermediate data. The *causal chain* and *causality* $< x_i, x_{i+1}, ..., x_j, y_i >$ satisfy dependency. For each causality in the causal chain, there exists a new causal chain satisfying its dependency, or the causality can be mapped to a precise mathematical formula or algorithm.

Domain knowledge of airborne software can be expressed as $< x_1, ..., x_n, y_m >$ by mathematical formula or algorithm. x_k and y_m are concepts. $< x_1, ..., x_n, y_m >$ represents that $< x_1, ..., x_n >$ has relationship with y_m. $< x_1, ..., x_n, y_m >$ is a knowledge unit satisfying causality, that is we can get y_m from $x_1, ..., x_n$ through mathematical formula or algorithm. In airborne software domain knowledge, the attribute of things can be expressed as a concept and the relation between concepts can be expressed as a relationship. In this way, airborne software domain knowledge can be formed into knowledge graph.

Definition 4: We introduce the triple N=(C,F,R) known as a directed graph. Where C represents knowledge concept set and its attributes such as performance, safety and security. F represents rule such as mathematical formula or algorithm. R represents the knowledge relationship. The structure is shown in Figure 1.

For example, velocity $v = at$, mileage $s = vt$. Then the concept v is generated by the acceleration concept a and time concept t, the concept s can be generated by concept v and time concept t. These based on causal model.

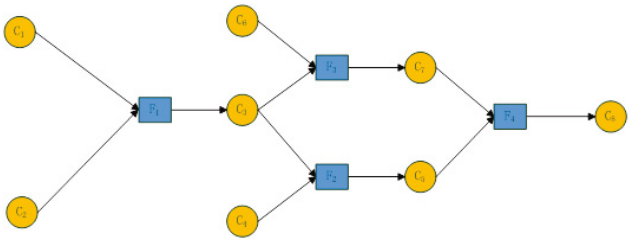

Fig. 1. Knowledge graph structure, Sequence $< C1, C2, F1, C3 >$ represents C3 is generated by C1 and C2 according to F1 and it is a relationship

Such model has the following properties:

1. Reachability. If you can reach a concept from initial concept, then the concept is reachable. N=(C,F,R), if there exists $f \in F$ makes $C[f] \rightarrow C_1$, then C_1 can be reached from C directly. If there exists a sequence $f_1, f_2, ..., f_{k-1}$ and $C_1, C_2, ..., C_k$ makes $C_1[r_1] \rightarrow C_2[r_2]...C_(k-1)[r_{k-1}] \rightarrow C_k$, then C_k can be reached from C_1.

2. No Deadlock. This model is established on the causality . Perform a sequence can ultimately lead to any other sequence, then it will not have deadlock.

3. Reversibility. Any concept reached from the initial concept can go back to the initial concept.

There are some advantages when adopt this model. First, model is described in a graphic way and it is easy to understand. At the same time, it supports mathematical analysis to improve the accuracy of semantic description. Second, the formal method can realize concurrency, synchronization, resource sharing modeling of the system although not discussed in this paper. Third, it is helpful to analysis and verify requirements. This can be analyzed in section 4.

Although the model represented by figure is intuitive, it is poor normative and not identified in other systems. Then we propose XML to represent domain knowledge model in order to make up the defects.

3.2 XML Representation Framework of Knowledge Graph

Our knowledge graph includes three model elements: concept, relationship and rule. The concept describes attributes of knowledge. It has an identified ID and Name, performance attributes, safety-related attributes and interface description. The specific model structure of concept is shown in the left of Figure 2. The relationship describes the relation between concepts. Each relationship has an identified ID and Name, inputs section Variables, outputs section DependentVariables and corresponding rule section Description. The specific model structure of relationship is shown in the right of Figure 2. The rule is mainly formulas. We adopt MathML tool to express it. Each rule corresponds to an XML document. An example is shown in Figure 3.

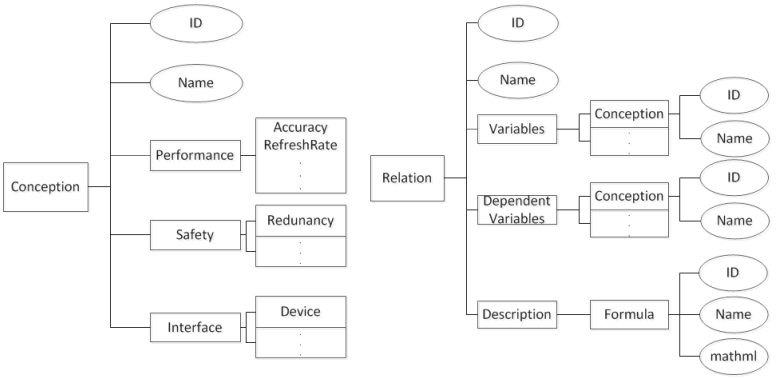

Fig. 2. Left is concept model structure, right is relationship model structure

```
<math>
  <mrow>
    <mfrac>
      <mrow>
        <mtext mathvariant='normal'>d</mtext>
        <mi>H</mi>
      </mrow>
      <mrow>
        <mtext mathvariant='normal'>d</mtext>
        <mi>t</mi>
      </mrow>
    </mfrac>
    <mrow>
      <mo>=</mo>
      <mi>v</mi>
    </mrow>
    <mtext mathvariant='normal'>d</mtext>
    <mo> ∀ </mo>
  </mrow>
</math>
```

Fig. 3. An example of the rule represented by MathML

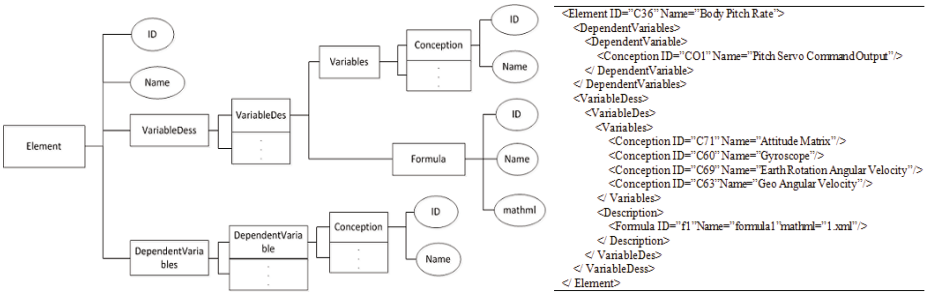

Fig. 4. Left is the complete domain knowledge graph model structure of airborne software, right is a detailed XML description of element 'Body Pitch Rate'

Finally, the complete knowledge graph model structure is shown in the left of figure 4. Element represents a knowledge. Each Element has an identified ID and Name. VariableDess represents a collection of knowledge and correspond rule which the knowledge depends. VariableDes represents single knowledge and correspond rule it depends. Variables represents knowledge which the knowledge depends. Conception represents a single knowledge. Formula represents correspond rule like mathematical formula or algorithm. DependentVariables represents a collection of knowledge which depends on the Element. DependentVariable represents single knowledge which depends on the Element. An example of specific knowledge graph XML description is shown in the right of Figure 4.

4 Formal Modeling of High-Level Requirements Based on Knowledge Graph(RMKG)

Due to the high safety and reliability features of airborne software, coupled with the importance of requirement analysis, high-level requirements modeling and verification is critical. The paper describes high-level requirements formally in order to verify conveniently and it generates high-level requirements by knowledge reasoning. Through an effective mechanism it converts non-functional requirements such as safety, security and real-time into related non-functional requirements of high-level requirements to ensure airborne software's safety and reliability at high-level.

4.1 Formal Description of High-Level Requirements

Definition 1: *High-level requirements representation* is a functional assertion structure formed by matching knowledge graph based on initial concept and terminate concept sets. It can be expressed as $< x_1, ..., x_n, r, y_m >$ formally and it means y_m is calculated by formula r based on the inputs $x_1, ..., x_n$. Single high-level requirements XML representation is shown in Figure 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<Function>
  <TaskOutput>
    <Conception ID="C22" Name="Dynamic Pressure"/>
  </TaskOutput>
  <TaskInput>
    <Conception ID="C1" Name="Total Pressure Input"/>
    <Conception ID="C2" Name="Impact Pressure Input"/>
  </TaskInput>
  <Rule>
    <Formula ID="f1" Name="formula1" mathml="1.xml"/>
  </Rule>
</Function>
```

Fig. 5. An example of XML description of a single high-level requirements

The TaskOutput indicates the generated knowledge, TaskInput represents knowledge inputs and Rule represents rules such as mathematical formulas and so on which the generated knowledge depends on. Each high-level requirements only has one clear and unified interpretation and it is unambiguous and not in conflict with others. Using our formal representation, High-level requirements possess accuracy, consistency and concurrency.

4.2 Generation of High-Level Requirements

After the knowledge graph data structure has been determined, domain experts build knowledge graph by input knowledge through modeling tool. When the knowledge graph is generated, find and match knowledge graph. We use knowledge reasoning techniques to get all paths sets from the initial inputs to terminate outputs. Then we get high-level requirements. Shown in Figure 6. We can see given the initial inputs C_1, C_2, C_3, terminate outputs C_8, C_9, C_10 and knowledge graph, we'll get high-level requirements $< C_1, F_1, C_4 >< C_4, F_6, C_8 >,< C_2, F_2, C_4 >< C_4, F_6, C_8 >,< C_1, F_1, C_4 >< C_4, F_7, C_9 >,< C_2, F_2, C_4 >< C_4, F_7, C_9 >,< C_2, F_3, C_5 >< C_5, F_7, C_9 >$ and $< C_3, F_4, C_5 >< C_5, F_7, C_9 >$. $< C_3, F_5, C_6 >$ means initial input C3 can't get terminate outputs. $< C_7, F_8, C_{10} >$ means terminate output C_{10} can't be derived from initial inputs. So they are not included into high-level requirements.

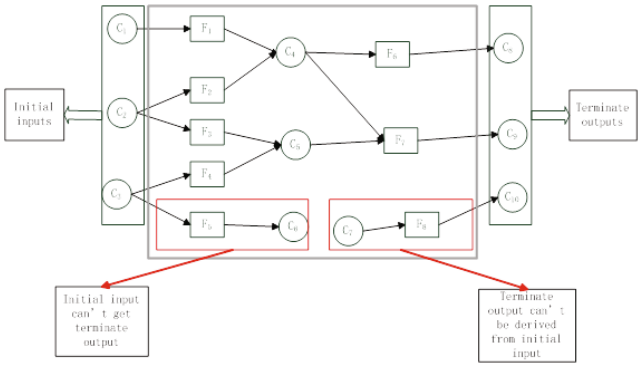

Fig. 6. High-level requirements formation process according to initial inputs and terminate outputs in system requirements

Requirement conversion is querying and inferring knowledge graph to obtain high-level requirements when given initial inputs and terminal outputs in the system requirements. We first use backward reasoning to get the causal chain of high-level requirements based on given outputs. For the redundant inputs we apply forward reasoning to achieve the causal chain of high-level requirements. Then we get all high-level requirements sets when given system requirements. On the one hand we reuse domain knowledge by querying and inferring mechanism.

On the other hand, we extend and improve knowledge graph while querying and inferring. Therefore, as the domain knowledge become diverse increasingly, the success rate of subsequent requirements matching can improve significantly.

We adopt improved breadth-first traversal in our backward and forward reasoning. According to our given complete knowledge graph model structure in section 3.2, our algorithm's variables correspond to the element name of the structure. *mapVariable* is the knowledge and its variabledess mapping. *mapdependVariable* is the knowledge and its dependvariables mapping. In backward reasoning, for each given output, we first judge that the dependent inputs variabledess of given output are all initial inputs or initial inputs and intermediate nodes or all intermediate nodes. If they are initial inputs, then we determine whether they are in given inputs, if they are intermediate nodes, we will recursively call backward reasoning algorithm. The backward reasoning algorithm is as follows.

Algorithm 1. BackwardTraversal Algorithm

```
Input ⇐ outputknowledge, variabledess
Output ⇐ stack
for each variables ∈ variabledess do
   if judge(variables)==1 then
      if IsInGivenInput(variables) then
         addToStack(variables)
         continue
      end if
   else if judge(variables)==2 then
      for each variable in variables do
         if IsInitialInput(variable) then
            if IsInGivenInput(variable) then
               addToStack(variable)
            end if
         else
            addToStack(variable)
            BackwardTraversal(variable,mapVariable[variable],stack)
         end if
      end for
   else
      for each variable ∈ variables do
         addToStack(variable)
         BackwardTraversal(variable,mapVariable[variable],stack)
      end for
   end if
end for
```

In forward reasoning, for each remaining input and its dependent output sets dependvariables which is a collection of arrays, we get the knowledge variabledess for each element conception in array of dependvariable, an element in dependvariables. For each element variable in array variables which is an the element

in variabledess, we judge whether it is an intermediate node or an initial input. If it is an intermediate node and not in the intermediate nodes which backward reasoning generates, we call BackwardTraversal algorithm. If it is an initial input and not in the given inputs, then the path is interrupted. If all the element in array variables are in given inputs or in the generated nodes, we add it to stack. Then judge whether the conception is final output or not. If conception is not final output, we will recursively call forward reasoning algorithm. The forward reasoning algorithm is as follows.

Algorithm 2. ForwardTraversal Algorithm

```
1: Input ⇐ inputknowledge, dependvariables
2: Output ⇐ stack
3: for each dependvariable ∈ dependvariables do
4:    for each conception ∈ dependvariable do
5:       variabledess = mapVariable[conception];
6:       for each variables ∈ variabledess do
7:          for each variable ∈ variables do
8:             if IsIntermediateNode(variable)&&!InGeneratedNodes(variable) then
9:                BackwardTraversal(variable,mapVariable[variable],stack)
10:            else if IsInitialInput(variable)&&!InGivenInput(variable) then
11:               break
12:            end if
13:         end for
14:         if IsAll(variables) then
15:            addToStack(variables)
16:         end if
17:      end for
18:      arr = mapdependVariable[conception]
19:      if arr!=null or arr.length!=0 then
20:         ForwardTraversal(conception,arr,stack);
21:      end if
22:   end for
23: end for
```

The improved breadth-first traversal can get all high-level requirements from system requirements. Because of reachability of our knowledge graph model, we can judge whether the initial inputs can reach the terminate outputs. As the model is alive, there's no dead loop in our algorithms. The model is reversible, so we can use backward reasoning to judge whether the terminate outputs can be got from initial inputs.

High safety and reliability is significant to airborne software. As the performance, safety and other attributes of concepts in knowledge graph may not meet the demands proposed in system requirements, We propose a reliable conversion mechanism to transfer non-functional requirements such as safety, security and real-time of system requirements into corresponding non-functional high-level requirements.

Non-functional Requirements Conversion Mechanism: For airborne software safety, assign the highest safety level to each node of high-level requirements according to output safety level given by system requirements. Namely, if the highest safety level of the given output is 5, then the safety level of each node in the high-level requirements is 5. Only when the safety level of each node in the high-level requirements is 5, the safety level of output we obtain can be 5. For airborne software security, assign the highest security level to each node of high-level requirements according to input security level given by system requirements. Namely, if the highest security level of the given input is 7, then the security level of each node in the high-level requirements is 7. Because the security level of input is the highest, then the security level of each node in the high-level requirements generated based on the input must be the highest. For airborne software real-time, assign the highest real-time to each node of high-level requirements according to system requirement. If the highest real-time in system requirements is sample once every 10ms, then each node of the high-level requirements also sample once every 10ms. Only in this way can we get the highest real-time system requirements.

Our conversion mechanism allocate safety, security and real-time according to the highest level, so the high-level requirements we get own the highest safety, security and real-time. Ensure the related requirements such as safety, security and real-time of airborne software are developed into high-level requirements, we start to monitor non-functional requirements from high-level requirements, which plays an important role in forming high safe and reliable airborne software finally.

5 Experimental Results

In this section, we present some experiments. The knowledge graph is built based on atmospheric data calculation, flight control and flight management which airborne software use. There are 73 concepts, 37 relationships and 47 mathematical formulas. We provide modeling tool for domain experts to construct knowledge graph collaboratively. When domain experts input xml files of concepts or relations, knowledge graph built according to the input xml files. If they search knowledge, the knowledge graph of the relevant knowledge will be shown. When expert modifies the knowledge graph, other experts will receive the modified message in the Latest News column. Experts can also decide whether to modify the knowledge by online conversation, you can see in the Figure 7.

When experts input system requirements files, the system will presents a graphical display of high-level requirements to experts. Experts can see how system requirements are turned into high-level requirements clearly. They will eventually get written documents of high-level requirements. We give some verification information from the aspect of integrity, consistency, correctness and traceability. When conflicts are detected among the graphs, the source can be identified and the problem can be clearly specified. Some verification information is given in the Latest News column finally. We give two examples in the Figure 8, when input different system requirements, the results are different.

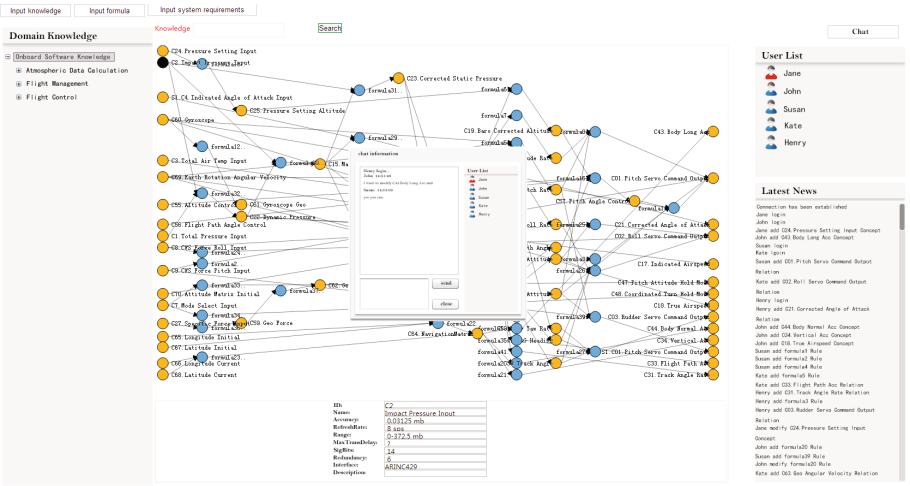

Fig. 7. An example of collaboration based on online conversation

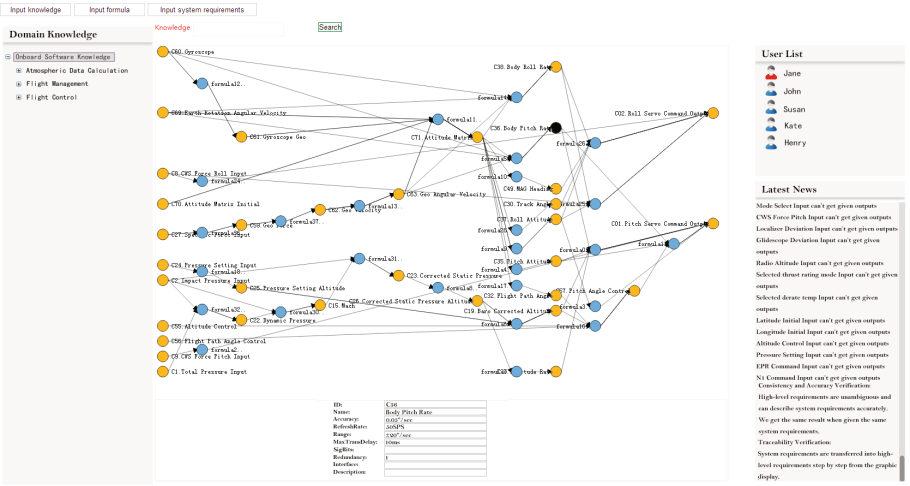

Fig. 8. Examples of high-level requirements formation and verification

6 Conclusions and Future Work

In this paper we have presented a new approach of high-level requirements modeling of airborne software. Formal description of knowledge graph is helpful to analysis and verify requirements. Formal description of high-level requirements makes it have accuracy, consistency and concurrency. Ensure the safety, security and real-time of airborne software at high-level requirements makes it important to form high safety and reliability software. We realize a modeling tool for experts to construct knowledge graph collaboratively, and complete the conversion from system requirements to high-level requirements based on knowledge graph. Our tool realize traceability automatically which can only be achieved manually in traditional methods and we obtain pretty good results in our experiments. Further, we firstly do more theoretical research. Moreover, we will improve our experiment to support the following airborne software architecture design.

References

1. Camus, J.L.: The Airborne Software Development Challenge. White Paper, Esterel Technologies (2010)
2. RTCA Inc.:RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Washington D.C(1992)
3. RTCA Inc.:RTCA/DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Washington D.C(2011)
4. Zhong, S.W., Hong, M.E.I.: Review of the unified modeling language (UML). Journal of Computer Research and Development (1999)
5. Sowa, J.F.: Conceptual graphs as a universal knowledge representation. In: Computers & Mathematics with Applications, pp. 75–93 (1992)
6. Hoang, N.V., Valerie, G.B., Marta, R.: Object detection and localization using a knowledge graph on spatial relationships. In: 2013 IEEE International Conference on Multimedia and Expo (2013)
7. Delugach, H.S., Brian, E.L.: Acquiring software requirements as conceptual graphs. In: Fifth IEEE International Symposium on Requirements Engineering (2001)
8. David, S.: AADL Display System Model Description. Rockwell Collins, Inc. (2004)
9. OMG:UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.0 (2009)
10. Bouquet, P.: Theories and uses of context in knowledge representation and reasoning. Journal of Pragmatics, 455–484 (2003)
11. Yao, Y.L.: A Petri net model for temporal knowledge representation and reasoning. IEEE Transactions on Systems, Man and Cybernetics, 1374–1382 (1994)
12. Chen, S.M.: Fuzzy backward reasoning using fuzzy Petri nets. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 846–856 (2000)