International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(3): 263–286, doi:10.21655/ijsi.1673-7288.00260

Research Article

Approach to Generating TAP Rules in IoT Systems Based on Environment Modeling

Han Bian (边寒)[1], Xiaohong Chen (陈小红)[1], Zhi Jin (金芝)[2,3], Min Zhang (张民)[1]

[1] (Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China)
[2] (Department of Computer Science and Technology, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)
[3] (Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)
Corresponding author: Xiaohong Chen, xhchen@sei.edu.cn; Zhi Jin, zhijin@pku.edu.cn

Abstract User requirements are the fundamental driving force of smart services in Internet of Things (IoT). Today, many IoT frameworks such as IFTTT allow end users to use simple Trigger-Action Programming (TAP) rules for programming. However, these rules describe device scheduling instructions instead of user service requirements. Some IoT systems propose goal-oriented requirement approaches to support service goal decomposition. Nevertheless, it is difficult to ensure the consistency of different services and completeness of service deployment. To achieve correct "user programming" in IoT systems and ensure the consistency and completeness of user service requirements, this study proposes an environment modeling-based approach to automatically generate TAP rules. On the basis of the service requirements provided by users, required system behaviors are automatically extracted according to the environment model. After their consistency and completeness are checked, TAP rules are generated, which realizes automatic generation from user service requirements to device scheduling instructions. The environment ontology of IoT application scenarios is constructed for environment modeling, and the description method of service requirements based on the environment ontology is also defined. Finally, the accuracy, efficiency, performance of the approach, and the time cost for building the environment ontology are evaluated with a smart home scenario. The results show that the accuracy, efficiency, and performance of this approach exceed the available threshold, and the time cost in building the environment ontology can be ignored when the number of requirements reaches a certain level.

Keywords Internet of Things (IoT); user service requirements; requirements consistency; requirements completeness; environment modeling

Citation Bian H, Chen XH, Jin Z, Zhang M. Approach to generating TAP rules in IoT systems based on environment modeling, *International Journal of Software and Informatics*, 2021, 11(3): 263–286. http://www.ijsi.org/1673-7288/260.htm

---

This is the English version of the Chinese article "基于环境建模的物联网系统 TAP 规则生成方法. 软件学报, 2021, 32(4): 934–952. doi: 10.13328/j.cnki.jos.006224".
Funding items: National Key Research and Development Program of China (2018YFB2101300); National Natural Science Foundation of China (61620106007, 61751210, 61872146); Science and Technology Project of Shanghai Municipality (20ZR1416000)
Received 2020-09-13; Revised 2020-10-26; Accepted 2020-12-19; IJSI published online 2021-09-25

Internet of Things (IoT) systems monitor, schedule, and manage various devices, such as cars, traffic lights, air conditioners, and bulbs, through sensors and actuators connected by the network, so as to provide users with a variety of smart services to meet their service requirements and facilitate users' daily life. In such a system, user service requirements are the fundamental driving force of smart services. In recent years, user-oriented programming frameworks for the IoT, such as IFTTT and Microsoft Flow[1], are the products of the rapid development of IoT. They allow users to program with the simple rule "IF $trigger$, THEN $action$", called Trigger-Action Programming (TAP). Users can make the following rule in the smart home system shown in Figure 1: "If the light brightness exceeds 50,000 lx, the blind will be closed" and this rule can be described by TAP as "IF $Light.brightness >$ 50,000 lx THEN close the blind". However, this TAP syntax in fact describes a device scheduling instruction. Telling the software to close the blind is neither a user service requirement nor a system behavior. The service requirements are close to the user needs. For example, users want to close the blind in order to make the light dim. System behaviors are close to software and care about events, such as sending a pulse to close the blind. The "action" in TAP syntax refers to the instruction informing the software to close the blind, which is different from the state "the closed blind" or the event "the pulse to close the blind".

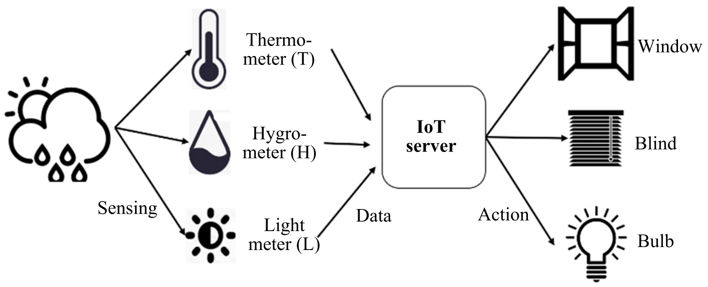


Figure 1 An example of smart home

Some IoT systems propose to adopt the goal-oriented requirements approach. For example, Reggio uses the domain model and the UML diagram to capture service requirements and support the service goal decomposition [2]. However, the author has not proposed specific measures to guarantee the consistency among different IoT services and the completeness of service deployment, which is prone to cause problems in consistency and completeness. The problems in consistency refer to the conflict between two or more service requirements[3, 4], while the problems in completeness refer to the situations in which the software is unable to give exact outputs for the inputs in some cases[5]. For the software system, any error caused by inconsistency and incompleteness may lead to adverse consequences in the real world, and even threaten the property and life of users[6]. In an IoT system, users do not consider the problems of consistency and completeness when using TAP rules for programming, which makes these problems more prominent. For example, in a smart home software system, if users set the window to automatically open when the concentration of carbon monoxide is above a threshold and automatically close the window when it rains, a consistency problem will occur when the concentration of carbon monoxide is too high during rain. In this case, the concentration of carbon monoxide in the room may be more than 200 mg/L, which is harmful or even fatal. As another example, users specify the bulb to be on when the brightness is below a threshold, but do not specify the behavior above this threshold; in this case there will be a completeness problem, which makes the bulb always on and results in a serious waste of energy. Plenty of studies provide the checking and repairing approaches regarding the consistency of TAP rules[7, 8], but there is no checking at the service requirement and system behavior levels.

To realize the correct "user programming", this paper proposes to automatically extract system behaviors depending on the service requirements provided by users. After consistency and completeness have been checked, system behaviors can be automatically converted into device scheduling instructions in the form of TAP. With regard to defining the service requirements, on the basis of "Environment Modeling based Requirements Engineering"[9, 10], we think that it is more appropriate to represent the service requirements with the state changes of devices than TAP. For example, users want the bulbs to be off, but they do not care what kind of behaviors makes the bulbs off. According to Reference [11], the derivation process from the service requirements to the system behaviors must take into account environmental characteristics. Consistency and completeness should be checked from the perspective of the device. Therefore, this paper proposes an approach to generating the TAP rules in IoT systems based on environment modeling and achieves the automatic generation from service requirements to programs. The main contributions of this paper are as follows:

(1) The environment ontology of IoT application scenarios is constructed, and the basic concepts and their relations of environment modeling are provided. According to the environmental characteristics of IoT application scenarios, the monitored entity and the controlled entity are defined, and the corresponding characteristics are modeled with attributes and state machines.

(2) The definition of environment based user service requirements is given. The service requirements are denoted as state changes. Additionally, the description method of user service requirements based on environment ontology is defined.

(3) The rules of consistency and completeness of the requirements of the IoT systems are defined. On the basis of the environment ontology, the problem diagram[12] is used to carry out the automatic transformation from service requirements to system behaviors, and the consistency and completeness checking of the requirements is completed in line with the rules.

The problem diagram used in the derivation of the system behaviors is introduced in Section 1. The environment ontology of IoT systems is defined in Section 2. The methodology of this paper is presented in Section 3. The key algorithms are shown in Section 4. Experiments to evaluate the method proposed in this paper are designed in Section 5. Related work is compared in Section 6. Conclusions are summarized and some future work is put forward in Section 7.

1 Preliminaries

A problem diagram is the result of the requirements description of the problem frame approach[12], which is used in this paper to represent user service requirements and software behaviors. Figure 2 shows a simple example of a problem diagram. The software problem is to specify a system to be developed (controller machine) to monitor and control the problem domain (room and air conditioner) so as to meet requirements (adjust temperature). The connection between the problem domain and the machine is called an interface (an interaction), which indicates the phenomena shared between them, such as the *OnPulse* and *OffPulse* shared between the controller machine and the air conditioner. The interface is initiated by one problem domain or the machine and it is denoted by "`Initiator!{content}`". The content can be an event, a state, or a value in the phenomenon. A requirement is expressed as an expectation described in natural language, which is actually denoted as changes that are expected to occur in the problem domain, such as the room temperature $T > 30$ and the air conditioner being turned *on* or *off*. This expectation is called requirement phenomenon. The references to the requirement phenomena that can only be observed but not be controlled are called **requirement**

references. For example, the room temperature can only be observed and referenced, but cannot be controlled. Requirement phenomena that can be controlled are called **requirement constraints**. For example, an air conditioner can be switched on and off. Both requirements references and constraints can be expressed as interface interactions.

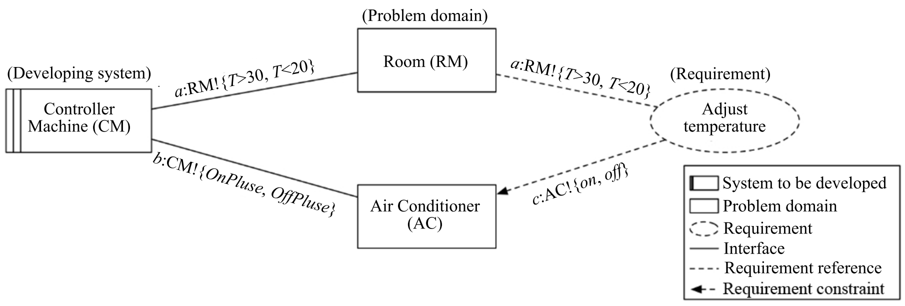


Figure 2 Simple example of a problem diagram

According to Reference [12], a problem diagram can be represented by a six-tuple $\mathit{ProDgm} \overset{\text{def}}{=} \langle M, \mathit{Pds}, \mathit{Reqs}, \mathit{Int}, \mathit{Ref}, \mathit{Con}\rangle$, where M, *Pds*, *Reqs*, *Int*, *Ref*, and *Con* represent the controller machine (system to be developed), the set of problem domains, the set of requirements, the set of interfaces, the set of requirement references, and the set of requirement constraints, respectively.

The two sides of the problem diagram represent user service requirements and software behaviors respectively. The user service requirements are described in terms of requirement phenomena and their relationships, and presented at the requirement references and constraints on the right side of the problem diagram. The requirement phenomena in a and c on the dashed line on the right of Figure 2 require that the air conditioner should be turned on and off when the temperature is higher than 30°C and lower than 20°C, respectively, which is the service requirement of the user. Software behaviors are described in terms of specification phenomena and their relationships and presented at the interfaces on the left side of the problem diagram. In interfaces a and b represented by solid lines, the sharing phenomena require that the controller should send out the pulse to turn on the air conditioner when the temperature is higher than 30°C and the pulse to turn off the air conditioner when the temperature is lower than 20°C, which defines the system behaviors of the software. It should be noted that deriving software system behaviors from user service requirements in the problem diagram requires the knowledge of the domain, which is described by an environment ontology in this paper.

2 Environment Ontology of IoT Systems

Ontology is an explicit formal specification of a shared conceptual model[13]. We define the environment ontology to provide the basic concepts of environment modeling and the relationships between them. According to the domain-related condition, the environment ontology can be divided into two categories: upper environment ontology and domain environment ontology. The upper environment ontology describes the concepts and associations in general environment modeling, while the domain environment ontology is the result of environment modeling for a specific domain, which is the instantiation of the concepts and associations of the upper environment ontology in a specific field.

2.1 Upper environment ontology

The environment of an IoT system can be regarded as a set of entities (called environment entities) that interact with the system. Therefore, the environment entity is the most basic concept in the environment ontology. In specific application scenarios entities such as smart homes, entities like the bulb, blind, people, air. are all environment entities. These environment entities can be divided into two categories.

(1) **Monitored entity**: The state value of such an environment entity can only be obtained by monitoring the IoT system, but cannot be directly changed, which means that the state of the entities changes autonomously and is independent of the will of human beings. For example, indoor air and brightness involved in a smart home system belong to monitored entities.

(2) **Controlled entity**: Regarding this kind of environment entities, the state value (namely the value of each attribute) can be obtained directly and the pre-defined instructions can be sent to change the entity's state. A variety of embedded devices fall into this category, such as fluorescent lamps, windows, and electric curtains in a smart home system. Either the state or the value of the attribute can be changed. For example, lamps can be described by the state "*on*" or "*off*" or by a specific value of $brightness$.

Attributes of each environment entity can be used to describe its properties, and each attribute has a value. For example, the light of an environment entity to be monitored has the brightness attribute represented by $Light.brightness$, whose value represents the brightness of the light. Another example is the controlled environment entity *Bulb*, whose attribute is *BulbST*, namely the state of the bulb. The attribute value can be *bon* or *boff*, representing the on-off state of the bulb. In addition to the attributes, state machines can also be used to describe the dynamic characteristics of some controlled entities, involving states and transitions. Transitions are triggered by events. For example, a state machine with state *bon* or *boff* can be used to describe the dynamic characteristics of a bulb. The state changes to *bon* when the bulb receives the *bonPulse* action, while the state changes to *boff* when it receives the *boffPulse* action.

There are also some associations between these concepts, as shown in Figure 3, and the meaning of these associations is listed in Table 1.

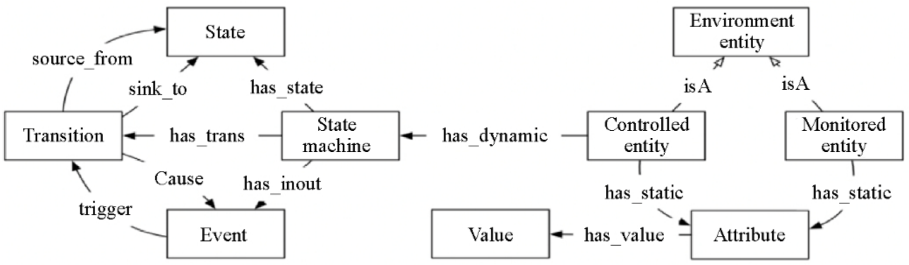


Figure 3 Concept association diagram of the environment ontology

2.2 Domain environment ontology of a smart home

With the smart home as an example, the construction of the domain environment ontology is illustrated. The environment entities in the domain are identified firstly. Window, blind, and bulb are assumed to be the controlled environment entities in the smart home domain, and the monitored environment entities are light and air. These are the instances of environment entities. The attribute of the window is *WindowST*, which describes the opening and closing of the window, with the value of *wopen* or *wclosed*. The attribute of the blind is *BlindST*, which describes the opening and closing of the blind, with the value of *bopen* or *bclosed*. The attribute of the bulb is *BulbST*, which describes the switch of the bulb, with the value of *bon* or *boff*. The attribute of light is $brightness$, which means the brightness, and its value could theoretically be any value greater than 0. The attributes of the room include the temperature and the humidity. The variate $temperature$ represents the temperature from -273.15°C to infinity. The variate $humidity$ refers to the relative humidity of the air, which can theoretically range from 0% to 100%. These specific attributes and values are respectively the instances of these two concepts.

Table 1 Associations and meanings of concepts in environment ontology

Name	Representation	Meaning
has_static	Environment Entity → Attribute	Each environment entity has multiple attributes
has_dynamic	Controlled entity → State machine	Each controlled entity has multiple state machines
has_value	Attribute → Value	Each attribute has a set of values
has_state	State machine → State	Each state machine has multiple states
has_trans	State machine → Transition	Each state machine has multiple transitions
has_inout	State machine → Event	Each state machine has multiple input/output events
source_from	Transition → State	Each transition results from a state
sink_to	Transition → State	Each transition ends in a state
cause	Transition → Event	Each transition triggers multiple events
trigger	Event → Transition	Each event triggers multiple transitions

The state machine of each controlled entity needs to be further identified. The state machines of window, blind, and bulb in the context of a smart home are shown in Figure 4. With the bulb as an example, the *bon* state means that the bulb is on. If the bulb receives a *bonPulse* event at this time, it will continue to be on; if the bulb receives a *boffPulse* event, it will be off. When it is in the *boff* state, the bulb is off. If a *bonPulse* event is received at this time, the bulb becomes on; otherwise, if a *boffPulse* event is received, the bulb is still off. These are all specific instances of the concepts of state machine, event, state, and transition.

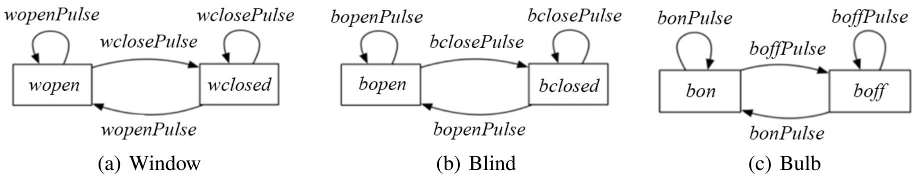


Figure 4 State machines of the controlled environment ontology for smart home systems

3 Approach for Generating TAP Rules

With the support of the environment ontology, this paper proposes the approach for generating TAP rules (Figure 5). After users write the service requirements, system behaviors are firstly derived with the help of the environment ontology, after which the initial problem diagram is obtained. Then the consistency and completeness of the problem diagram are checked. If it passes the checking, the checked problem diagram will be combined with the phenomenon-instruction look-up table to generate TAP rules. Otherwise, the problem diagram will be returned to the users to modify the service requirements.

3.1 Writing of user service requirements

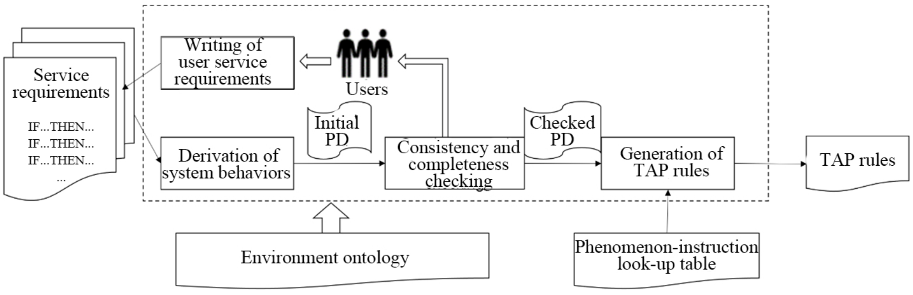
First, we define the user service requirements. With the idea of environment modeling, this paper denotes the service requirements as the changes of environment entities. In IoT systems, the changes of environment entities are usually represented by different states of the environment entities. For example, the window should be kept closed when it rains. According to the "IF $trigger$ THEN $action$" syntax of the TAP rules, the following syntax is used in this paper to allow users to express service requirements.

$$\text{"IF } \langle entity.trigger \rangle \text{ THEN } \langle entity.state \rangle\text{"}$$

where

(1) *entity.trigger* can be expressed as follows.

- When the entity is a monitored environment entity, $entity.attribute$ is used to indicate that the attribute of the entity takes a certain value or is located in a certain range. For example, the brightness of light at 3,000 lx is written as $Light.brightness = 3{,}000$ while the brightness of light greater than 3,000 lx is written as $Light.brightness > 3{,}000$ (for the convenience of the automation process, only values are used while units are omitted).
- When the entity is a controlled environment entity, $entity.state$ can be used to indicate that the entity is in a certain state, e.g., *Window.wclosed* indicates that the window is closed. We can use $entity.attribute$ as in the previous case.
- In addition, the trigger can be a complex sentence using Boolean operators like "`and`" (written &&) and "`or`" (written ||).



Service requirements
IF...THEN...
IF...THEN...
IF...THEN...
...
Writing of user service requirements
Users
Initial PD
Derivation of system behaviors
Consistency and completeness checking
Checked PD
Generation of TAP rules
TAP rules
Environment ontology
Phenomenon-instruction look-up table

Figure 5 Architecture of our approach

(2) The $entity.state$ refers to the controlled environment entity and its state.

For example, if a user wants to express that the blind is closed when the brightness of light is greater than 50,000 lx, the entity in the trigger condition is the light, which is the monitored entity. The brightness of light is greater than 50,000 lx means the attribute "*brightness*" of light is greater than 50,000 lx. Thus the $entity.trigger$ of this service requirement is $Light.brightness > 50{,}000$. To express that the blind is in the closed state, the $entity.state$ is *Blind.bclosed*, so this service requirement becomes "IF $Light.brightness > 50000$ THEN *Blind.bclosed*". If the user wants to express the service requirement that the bulb is turned on when the blind is closed, the entity is the blind, which is the controlled entity. The trigger, i.e., the closed state of the blind, is *bclosed*. Therefore, the $entity.trigger$ of this service requirement is *Blind.bclosed*. To indicate that the bulb is on, the $entity.state$ is *Bulb.bon*, and thus this service requirement is "IF *Blind.bclosed* THEN *Bulb.bon*".

It is the same for applying to complex service requirements with the keyword "`and`" or "`or`". For example, to express that the window is open when the temperature is higher than 25°C and the brightness is greater than 25,000 lx, this service requirement has two trigger conditions and involves two monitored entities, i.e., air and light. The two trigger conditions are the temperature attribute of the air being greater than 25°C and the brightness attribute of the light being greater than 25,000 lx. Therefore, the first half of this service requirement is the "and" of *Air.temperature* > 25 and $Light.brightness > 25{,}000$, while the second half, i.e., the $entity.state$ is the open state of window, namely *Window.wopen*. Thus, this service requirement can be expressed as "IF *Air.temperature* > 25 && $Light.brightness > 25000$ THEN *Window.wopen*".

In addition, the environment ontology needs to be introduced when users write service requirements, so that users can choose the corresponding device, rather than having to guess what to write. The advantage of doing this is that most users are not familiar with the performance and effects of all devices and letting users write the requirements without constraints will create plenty of errors or omissions due to their lack of familiarity with the devices. The environment ontology contains different devices and their states and effects. The use of the environment ontology can remind users of the effects of the devices when they write service requirements; this helps in preventing users from writing some absurd service requirements, thus reducing the workload of the later checking and modification.

3.2 Derivation of system behaviors

System behaviors can be derived in two steps.

The first step is to annotate the problem diagram with the service requirements defined in Section 3.1. Each service requirement corresponds to a problem diagram, and the information of service requirement defines the right side of the problem diagram. The specific correspondence is as follows: First we have a correspondence between the environment entity and the problem domain. In "IF $\langle entity.trigger\rangle$ THEN $\langle entity.state\rangle$", *entity* represents the environment entities interacting with the system, namely the problem domains in the problem diagram. For example, *Room* and *Window* in the service requirement "IF *Room.temperature* > 30 THEN *Window.wopen*" shown in Figure 6 can be directly transformed into problem domains *Air* and *Window*. Then we have the correspondence between trigger/state and the phenomena in requirement references and constraints. A trigger, as the condition of requirements, represents the reference to requirements and thus can be drawn directly as requirement reference. A state is the phenomenon that users expect to see. It is the constraint on the requirements and can be directly drawn as requirement constraint. The phenomenon initiator should be the entity owning the state. For instance, *Room.temperature* > 30 in Figure 6 can be directly transformed into the requirement reference *Room*!{*temperature* > 30}, and *Window.wopen* can be directly transformed into the requirement constraint *Window*!{*wopen*}. The requirements drawn in an ellipse can be labelled directly, such as *Req*1 shown in Figure 6.

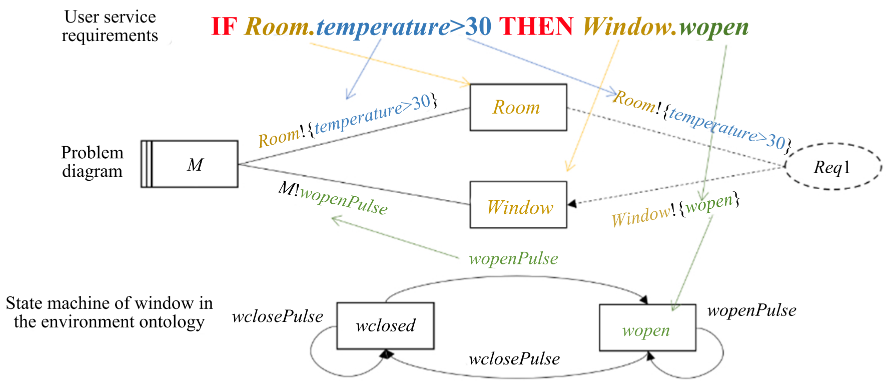


Figure 6 Schematic diagram of the problem diagram generation

The second step is to derive the corresponding system behaviors according to the environment ontology, namely defining the interface on the left of the problem diagram. For simple service requirements without the keywords "`and`" and "`or`" in the $entity.trigger$, they can be handled separately depending on the different entity types of triggers and states.

- No matter whether the entity in $entity.trigger$ is a monitored or controlled entity, the conditions in the requirement reference are copied directly, just like the interface between M and Air in Figure 6.
- For the entity in $entity.state$, the event that triggers the state should be found out through its state machine and put in the interface as a sharing phenomenon, and this interface must be sent over by the software. As shown in Figure 6, the trigger event of the state *wopen* in the *Window* state machine is *wopenPulse*, and thus the interface can be defined as $M!\{$*wopenPulse*$\}$.

The processing of complex user service requirements involving keywords "`and`" and "`or`" is described below.

- If multiple $entity.triggers$ are connected with "`and`", for each $entity.trigger$, the problem domains corresponding to the entity should be connected with the requirement through the requirement reference and the requirement phenomena are added on the requirement reference according to trigger.
- If they are connected with "`or`", namely that the requirement is in the form of "IF $a||b$ THEN $entity.state$", it means that if either a or b occurs, the entity should be in state $entity.state$. Since a and b are separated, the requirements become "IF a THEN $entity.state$" and "IF b THEN $entity.state$", which represent the same meaning. For example, "IF *Room.temperature* $<$ 25 || *Room.humidity* $>$ 25 THEN *Window.wclosed*" can be devided into "IF *Room.temperature* $<$ 25 THEN *Window.wclosed*" and "IF *Room.humidity* $>$ 25 THEN *Window.wclosed*". Therefore, the requirement with "`or`" should be divided into two sentences and respectively transformed into problem diagrams.

3.3 Consistency and completeness checking

We define inconsistency and incompleteness from the perspective of the environment entities. On the basis of the conventional inconsistency definition of the service requirements, in this paper we consider three types of requirement inconsistency according to the different cases of trigger and state.

(1) **Scope inconsistency**: The scopes in the triggers of more than one requirement have an intersection, but their state entities are the same and the contents are different, which results in conflicts between the requirements. For example, users of a smart home system want: "IF *Light.brightness* $<$ 30000 THEN *Window.wclosed*" and "IF *Light.brightness* $>$ 15000 THEN *Window.wopen*". These two requirements are applied to the window at the same time when the brightness of the light is between 15,000 lx and 30,000 lx. However, the states are opposite, which leads to the scope inconsistency.

(2) **Overwriting inconsistency**: The entities in states of multiple requirements are the same while the contents are diffirent. Hence, the state executed later overwrites the state executed first due to different triggers. For example, *Req*1 requires the bulb to be turned off when the brightness of the light is greater than 20,000 lx, while *Req*2 requires the bulb to be turned on when the blind is closed. If the blind is closed when the brightness of the light is greater than 20,000 lx, the bulb should be on at this point according to *Req*2, so the state *bon* overwrites the state *boff*. According to *Req*1, the bulb should be turned off since the brightness of the light is in fact still greater than 20,000 lx. As a result, the overwriting inconsistency appears.

Starting from the definition of the service requirements and depending on the nature of environment entities, this paper considers the following two cases regarding requirements incompleteness.

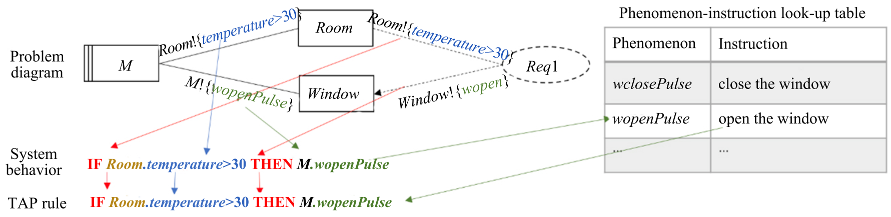
(1) **State incompleteness**: When the problem diagrams for all service requirements are combined, they do not involve all states of all entities in the environment ontology. This results in the requirements incompleteness. For example, there is only the state *Blind.bopen* in a set of requirements, without the state in which the blind is closed (i.e., *Blind.bclosed*), which is incomplete in state.

(2) **Attribute incompleteness**: The triggers of all requirements do not cover the full reachable range of the attributes of the corresponding entity, which makes the state of the entity not accurately determined in some cases. For example, when *Light.brightness* $>$ 20,000 and *Light.brightness* $<$ 15,000, the bulb should be off and on, respectively. However, when the brightness of the light is between 15,000 lx and 20,000 lx, should the state at the last moment be maintained or be changed in line with some other criteria? This is not mentioned in the service requirements of the bulb, and thus the requirement is incomplete in attribute.

3.4 Generation of TAP rules

The generation of TAP rules is divided into two stages.

(1) Generation of the system behaviors from the problem diagram: According to the checked problem diagram, whether the problem domain in the problem diagram corresponds to requirement reference or requirement constraint should be firstly determined. If it is corresponding to a requirement reference, the interface of the problem domain corresponds to the content of "IF". If there are multiple references, "&&" is used for connection. If it is corresponding to a requirement constraint, the interface of the problem domain corresponds to the content of "THEN". As shown in Figure 7, the problem domain *Air* is a requirement reference and its interface "*Room*!{*temperature* $>$ 30}" is the content of the "IF", while the problem domain *Window* corresponds to the requirement constraint and its interface "M!{*wopenPulse*}" is the content of "THEN". As a result, the system behavior "IF *Air.temperature* $>$ 30 THEN *M.wopenPulse*" is generated, but this is not the final TAP rule.



Phenomenon	Instruction
wclosePulse	close the window
wopenPulse	open the window
...	...

Figure 7 Schematic diagram of generating TAP rules

(2) Generation of TAP rules from system behaviors: The final TAP rule can be generated by the search of the system behaviors in terms of the relevant instructions in the phenomenon-instruction look-up table. The phenomenon-instruction look-up table shows the mappings between a set of software behaviors and instructions. As shown in Figure 7, *wopenPulse* corresponds to the instruction "open the window". This look-up table can be provided by experts in the device domain. After replacing "*M.wopenPulse*" in the system behavior by the instruction "open the window", the TAP rule "IF *Air.temperature* $>$ 30 THEN open the window" can be obtained.

4 Key Algorithms

To implement the method presented in Section 3, we develop a tool to generate the TAP rules, which can be accessed through `http://re4cps.org/dsigs` and the corresponding video demo is given at `http://re4cps.org/examples#DSIGS`. This section presents some key algorithms, including the algorithm of converting service requirements to problem diagrams, the consistency checking algorithm, the completeness checking algorithm, and the TAP generation algorithm.

4.1 Algorithm for transforming service requirements into problem diagrams

An algorithm for transforming service requirements into problem diagrams allows us to realize the derivation of the system behaviors mentioned in Section 3.2. The main idea of the algorithm is as follows. The "`and`" and "`or`" in the service requirements should be dealt with firstly. Then the problem domain and reference of the problem diagram are created for each service requirement according to the correspondence described in Section 3.2.

The specific steps are shown in Algorithm 1. The "`and`" and "`or`" in the service requirements are pre-processed in Line 5–Line 7; $entity.trigger$ of the requirement is traversed (if there is no "`and`" relationship connecting multiple $entity.triggers$, the loop is only carried out once) and the problem domain, requirement reference, and interface related to the trigger are constructed in Line 9–Line 13; the $entity.state$ is handled and the problem domains, requirement constraints, and interfaces related to the $entity.state$ are constructed in Line 14–Line 16. Assuming that the total number of service requirements is n, since there are two loops in the algorithm, which are both related to n, the time complexity of the algorithm is O(n^2).

4.2 Consistency and completeness checking algorithms

According to the definitions of consistency and completeness given in Section 3.3, we design the checking algorithms of requirements consistency and completeness. The main idea of the consistency checking algorithm is as follows: All (*trigger*, *state*) pairs are obtained according to the requirement references and constraints of the problem diagram firstly. Next, whether the triggers corresponding to the states of the same entity overlap is checked with the aim of checking the scope and overwriting inconsistency. The inconsistent requirements are fed back to the users. If there is inconsistency, the corresponding requirements will be fed back to the users.

The specific steps of the consistency checking algorithm are shown in Algorithm 2. The second line of the algorithm extracts (*trigger*, *state*) pairs from all problem diagrams, and the $entityMap$ is constructed in Line 3 to store the triggers corresponding to the states with the same entity but different contents. According to the $entityMap$, the loop from Line 4 to Line 16 checks the scope and overwriting inconsistency. If the inconsistency occurs during the checking process, the inconsistent requirements will be output and the algorithm will return false; if no inconsistency occurs, the algorithm will finally return true. There are three nested loops in the algorithm, but two loops are executed in constant times. Thus, the time complexity of Algorithm 2 is O(n).

To obtain all the triggers and the corresponding states contained in the requirement reference, this paper designs the *resolveProDgm* function to extract (*trigger*, *state*) pairs from all the problem diagrams. The specific steps are shown in Algorithm 3. Line 3 traverses the requirements in all the problem diagrams. The loop from Line 7 to Line 9 reads the phenomena in the requirement references and obtains the triggers, and Line 10 reads the phenomena in the requirement constraints and obtains the states. The number of requirements is assumed to be n

and thus the time complexity of the algorithm is O(n).

Algorithm 1. Service requirements are converted into problem diagrams.

```
Input: all service requirements $reqs$, the environment ontology $eo$;
Output: problem diagram set $ProDgms = \{prodgm_1, prodgm_2, \cdots, prodgm_n\}$.
1.  begin
2.  Initialize the integer $i = 1$, the problem diagram set $ProDgms$, and the machine $M$ of problem
      diagram; //initialization
3.  for $req \in reqs$ do   //convert each service requirement into a problem diagram
4.      Initialize each part of problem diagram $prodgm_i$, namely $Pds_i, Reqs_i, Int_i, Ref_i, Con_i$ //initialize
          each part of the problem diagram
5.      If there is no "and" and "or" relationship in $entity.trigger$ of req, $entity.trigger$ and
          $entity.state$ are read;
6.      If there is an "or" in $entity.trigger$ of req, the req is broken into multiple requirements and
          processed separately; //process "or"
7.      If there is an "and" in the $entity.trigger$ of req, a set of triggers will be constructed, and
          each $entity.trigger$ that is connected via "&&" is added to the set; //deal with the "and"
          relationship
8.      The requirement $Req_i$ in the problem diagram is constructed and added to $Reqs_i$; //construct
          the requirement in the problem diagram
9.      for $entity.trigger \in triggers$ do   //construct requirement reference and corresponding
          interface
10.         Construct the problem domain $pd$ according to the entity, and add $pd$ to $Pds_i$; //construct
              the problem domain according to the trigger
11.         Construct requirement reference $ref$ between $Req1$ and $pro$, add phenomenon to it
              according to the trigger, and add $ref$ to $Ref_i$; //construct requirement reference
12.         Construct an interface $int$ between $M$ and $pro$, add phenomenon to it according to the
              trigger, and add $int$ to $Int_i$; //construct the interface
13.     end for
14.     Construct the problem domain $pd'$ according to the entity in the $entity.state$ and add $pd'$ to
          $Pds_i$; //construct the problem domain according to $entity.state$
15.     Construct the requirement constraint $con$ between $Req_i$ and $pd'$, add phenomenon to it
          according to state, and add $con$ to $Con_i$; //construct requirement constraint
16.     Construct the interface $int'$ between $M$ and $pd'$, find the system behavior corresponding to
          the state in the $eo$, add it to the phenomenon of the interface, and add $int'$ to $Int_i$; //construct
          the interface to derive the system behavior
17.     Let the problem diagram be $prodgm_i = \{M, Pds_i, Reqs_i, Int_i, Ref_i, Con_i\}$, and add it to
          $ProDgms$; //construct the problem diagram
18.     $i$++;
19. end for
20. return $ProDgms$;
21. end
```

The main idea of the completeness checking algorithm is as follows: The (*trigger*, *state*) pairs in all the problem diagrams are extracted firstly. Next, all involved states are compared with the environment ontology for the checking of the state incompleteness, and the uninvolved states are fed back to the users. Finally, according to the triggers corresponding to the states of the same entity, the attribute incompleteness is checked and the uncovered cases are fed back to the users. The main steps are shown in Algorithm 4. The Line 2 of algorithm calls the *resolveProDgm* function and obtains all (*trigger*, *state*) pairs. The loop from Line 4 to Line 6 initializes the states. The Line 7–Line 10 check the state incompleteness with the states and the environment ontology. The loop of Line 11–Line 18 checks the attribute incompleteness. The number of requirements is assumed to be n. The Line 2 of Algorithm 4 uses the *resolveProDgm* function, whose time complexity is O(n). All subsequent loops have only one layer and run for n times at most. Thus, the time complexity of the algorithm is O(n).

Algorithm 2. Consistency checking algorithm.

```
Input: problem diagram set $ProDgms = \{prodgm_1, prodgm_2, \cdots, prodgm_n\}$;
Output: whether the consistency checking is passed and the inconsistent requirements.
1.  begin
2.  $map = resolveProDgm(ProDgms)$; //obtain all ($trigger$, $state$) pairs and store them in map
3.  Initialize a map $entityMap$ // $entityMap$ is a map. Its key is the names of controlled entities
      and its value is a list. Each element in the list is also a list, storing all triggers corresponding to one
      state of the key.
4.  for controlled_entity of environment ontology do
5.      Initialize a list $entityList$
6.      for $entityState \in$ all the states of controlled_entity do
7.          Initialize a list $tempList$
8.          for $(trigger, state) \in$ map do
9.              if $state == entityState$ then
10.                 $tempList.add(trigger)$
11.             end if
12.             $entityList.add(tempList)$
13.         end for
14.         $entityMap.put(controlled_entity, entityList)$
15.     end for
16. end for
17. return true; //there is no error, and thus return true
18. end
```

Algorithm 3. *resolveProDgm*: resolve problem diagram, and extract all triggers and states.

```
Input: problem diagram set $ProDgms = \{prodgm_1, prodgm_2, \cdots, prodgm_n\}$;
Output: all ($trigger, state$) pairs in $ProDgms$, which are stored in $map$.
1.  begin
2.  Obtain the reqirement of each problem diagram in $ProDgms$, and store it in $reqs$;
3.  for $req \in reqs$ do   //obtain trigger and corresponding state according to requirement references
      and constraints
4.      Initialize a character string set trigger and a character string variable state; //initialization
5.      Obtain problem domain linked with $req$; //obtain the entities in the corresponding requirements
6.      Respectively obtain the reference of $req$ in these problem domains, denoted as
          $ref_1, ref_2, \cdots, ref_{n-1}, con$; //obtain the carriers of trigger and state
7.      for $ref$ in $ref_1, \cdots, ref_{n-1}$ do   //obtain trigger
8.          Obtain the phenomenon of $ref$, and add this phenomenon to $trigger$; //obtain trigger
9.      end for
10.     Obtain the phenomenon $phe$ of $con$, and let $state = phe$; //find the state corresponding to the
          trigger
11.     Add ($trigger, state$) pair to $map$;
12. end for
13. return $map$;
14. end
```

4.3 Generation algorithm of TAP rules

According to the generation method of TAP rules described in Section 3.4, a specific generation algorithm can be designed. The main idea is as follows. Firstly, the content in the corresponding interface is referenced according to the requirement of the problem diagram to determine the trigger of TAP rule. After that, the corresponding interface is restrained according to the requirement of the problem diagram to construct the system behavior. Finally, according to the phenomenon-instruction look-up table, the corresponding instruction of the phenomenon is obtained to transform the system behavior into a TAP rule.

Algorithm 4. Completeness checking algorithm.

```
Input: problem diagram set $ProDgms = \{prodgm_1, prodgm_2, \cdots, prodgm_n\}$, environment
       ontology $eo$;
Output: whether completeness is satisfied; if not, output the reasons.
1.  begin
2.  $map = resolveProDgm(ProDgms)$; //call public functions
3.  Define linked list states to record the states involved in reqirements;
4.  for $(trigger, state) \in map$ do
5.      $states.add(state)$; //initialize states to check the incompleteness of the state
6.  end for
7.  if not all states are involved then //state incompleteness occurs
8.      Output the states not involved; //inform the user
9.      return false;
10. end if
11. Add the triggers corresponding to the states of the same entity in map into the same set, and add
      all this kind of sets into triggersList;
12. for $triggers \in triggersList$ do   //check the incompleteness attribute
13.     Compare the range of all triggers;
14.     if the range does not cover all the available values then   //attribute incompleteness occurs
15.         Output the values not covered; //inform the user
16.         return false;
17.     end if
18. end for
19. return true;
20. end
```

The main steps of the algorithm are shown in Algorithm 5. The Line 2 traverses all the checked problem diagrams. The Line 6 obtains the problem domains corresponding to all requirement references, and the loop from Line 7 to Line 10 obtains the phenomena in the interfaces corresponding to these problem domains and regards them as the triggers of the TAP rules. The Line 10 obtains the behaviors of the system and the Line 11 transforms the system behaviors into the corresponding instructions according to the phenomenon-instruction look-up table. Assuming that the number of requirements is n, the nested two loops, namely the loop in the Line 6, are executed in constant times. Therefore, the time complexity of the algorithm is O(n).

5 Evaluations

In the context of smart conference room systems, this section evaluates the proposed approach and answers the following questions.

(1) How are the accuracy and efficiency of this approach? Is it more accurate and efficient to write TAP rules using this approach than to write TAP rules manually?
(2) How is the performance of this approach? Is it suitable for large-scale systems?
(3) Does the construction time of environment ontology affect the use of this approach?

5.1 Accuracy and efficiency

To answer this question, we design a series of comparison experiments in this paper. First, the users are divided into two groups: Our approach Group (OG) and the TAP Group (TG). The OG first writes the service requirements, and then generates the TAP rules after checking and modifying the consistency and completeness with the proposed method. The TG writes the TAP rules manually and checks and modifies their consistency and completeness. To avoid the differences caused by user background, we further divide the users into two categories: professional users and non-professional users. Professional users are under graduate students majoring in software engineering, who have participated in some requirements related projects with certain knowledge of requirements engineering. Non-professional users are undergraduate students majoring in tourism economics, medicine, or linguistics, without any knowledge of computer or software engineering. On this basis, four experimental groups are set in total. Twenty-four users are invited, with six users in each group.

Algorithm 5. TAP generation algorithm.

```
Input: checked problem diagram set $ProDgms = \{prodgm_1, prodgm_2, \cdots, prodgm_n\}$,
       phenomenon-instruction look-up table;
Output: set of TAP rules.
1.  begin
2.  for $prodgm \in ProDgms$ do   //convert each problem diagram into a TAP rule
3.      Initialize the variables $triggers$ and $behavior$; //to store the content of IF and THEN in system
          behavior
4.      Let the requirement reference in prodgm be $refs = \{ref_1, ref_2, \cdots, ref_m\}$, and the requirement
          constraint be $con$; //obtain requirement reference and requirement constraint
5.      Let the problem domains linked to refs be $Pds = \{pd_1, pd_2, \cdots, pd_m\}$, and the problem
          domains linked to $con$ be $pd_c$; //obtain problem domains
6.      for $pd \in Pds$ do   //obtain the trigger of each TAP rule
7.          Let $int$ be the interface between $M$ and $pd$;
8.          Add the phenomenon of $int$ to $triggers$; if there are more than one phenomenon, "&&" is
              used for connection; //obtain triggers
9.      end for
10.     Let $int_c$ be the interface between $M$ and $pd_c$ and $behavior$ be the phenomenon of $int_c$; //obtain
          system behavior
11.     Let $action = table.get(behavior)$; //obtain the instructions in TAP rules according to
          phenomenon-instruction look-up table
12.     Add a character string (IF $triggers$ THEN $action$) in rules; //generate TAP rules
13. end for
14. return rules;
15. end
```

Each user is asked to write 10 requirements for a specific IoT scenario. The entities in the scenario only include air, light, people, window, blind, projector, air conditioner, and bulb. The air has two attributes (*temperature* and *humidity*). The light has one attribute (*brightness*). People can press the switch of the projector. The states of window and blind are *open* and *closed*, and the states of projector and bulb are *on* and *off*. The states of air conditioner can be *cold*, *hot*, and *off*. For the professional users of the proposed method group, we tell them the writing format of the requirements, the entities in the environment, and the state values of the entities. For the non-professional users of the proposed method group, because they do not have the domain knowledge, we directly tell them the available contents of $entity.trigger$ and $entity.state$ in the requirement pattern "IF $entity.trigger$ THEN $entity.state$". For the professional users of the TAP group, we tell them the format of the write rules, the entities in the environment, and the state values of the entities and ask them to speculate on the instructions in the environment ontology. For the non-professional users of the TAP group, we simply tell them the available contents of $entity.trigger$ and $action$ in TAP rule pattern "IF $entity.trigger$ THEN $action$". For the two kinds of users in the TAP group, we tell them the definitions of consistency and completeness after they write the rules and ask them to check and modify the rules they wrote manually.

In each group of experiments, the follwing statistics are made respectively: the time that the users in the OG spend on writing, the time that the users in the OG spend on checking and modifying the requirements to realize their consistency and completeness, the time that the users in the TG spend on writing rules, the time that the users in the TG spend on checking the consistency and completeness manually, the time that the users in the TG spend on modifying the rules, the number of consistency and completeness errors before and after user modification, and the accuracy of the final TAP rules obtained by all users. In the actual recording, since the two OGs use computer programs to automatically check the consistency and completeness, the checking time is very short, which is ignored in the total time. The final experimental results are shown in Table 2.

The accuracy and efficiency of this method do exceed the available thresholds (Table 2). First, according to the number of inconsistent and incomplete cases before and after modification, compared with the manual checking and modification of the TAP group, the checking and modification using the proposed method greatly reduce the number of errors. As can be observed from the last line of each group in the table, the accuracy of TAP rules obtained by both professional and non-professional users of the OG is generally higher than that obtained in the TG. Second, by comparing the first four lines of each group in the table, namely the time-related data, we find that the time that both professional and non-professional users of the OG spent on writing requirements is generally longer than the time that the TG spent on directly writing the rules, but after checking and modifying the consistency and completeness, the total time of the OG is basically shorter than that of the TG. Therefore, we can answer Question (1). Compared with the TAP rules written manually, those generated with the proposed approach are indeed more accurate and efficient.

Table 2 Result of user study

Items to be compared	Professional						Non-professional					
	OG			TG			OG			TG		
	Max	Min	Ave	Max	Min	Ave	Max	Min	Ave	Max	Min	Ave
Writing time (min)	30	19	25.0	23	9	15.8	30	15	22.8	32	17	21.7
Checking time	24 ms	9 ms	16.7 ms	13 min	5 min	9.8 min	31 ms	8 ms	18.2 ms	6 min	3 min	4.7 min
Modifying time (min)	20	5	11.5	24	5	15.8	6	2	3.7	17	11	14.0
Total time (min)	39	35	36.5	52	28	41.5	33	17	26.5	48	35	40.3
Inconsistency before modification	2	0	1.0	3	0	1.2	4	1	2.5	6	1	3.2
Incompleteness before modification	4	1	2.5	4	2	3.0	5	1	3.3	5	0	2.7
Inconsistency after modification	0	0	0.0	2	0	0.7	0	0	0.0	3	1	2.3
Incompleteness after modification	0	0	0.0	2	0	1.2	0	0	0.0	4	0	1.7
Final accuracy (%)	100.0	90.0	95.8	100.0	78.6	84.0	100.0	83.3	89.9	75.0	60.0	66.0

This approach is more useful for non-professional users. Both the total time and the final accuracy of the non-professional users in the OG are much better than those of the non-professional users in the TG. The reason is that professional users can achieve a relatively high accuracy in the manual consistency and completeness checking due to their rich background knowledge. Therefore, this phenomenon is not very distinct among them, but is very obvious among the non-professional users. As a result, we believe that the proposed approach is indeed more user-friendly to non-professional users compared with the manual processing.

We also note that there may be significant differences in writing time and final accuracy among different users in the same group, and there are two reasons for such differences.

Firstly, the complexity of requirements and rules are different, which results in differences in accuracy. For example, the trigger of requirements or rules written by some users neither contain the "`and`" and "`or`" connectors nor the states of the controlled entities, but only the attribute values of the monitored entities. Such requirements or rules are relatively simple and can be written with a high accuracy. However, other users are obsessed with complex requirements, which leads to more errors.

Secondly, differences in knowledge in other fields cause different writing time. For example, with the major of software engineering and a certain understanding of requirements engineering, a professional user spent only 12 min on writing the requirements, while another professional user spent 30 min. Through inquiry, we find that the latter has no ideas about the unit of light (lx) and the relative humidity, and thus more time is spent on checking the data to complete the writing of the requirements. Therefore, although there is not much difference in professional background, other factors such as life experience and knowledge in other fields can also lead to the fluctuations of the evaluation results.

Contrary to the popular belief, the final accuracy of some professional users fails to reach 100%, and the writing time of professional users is generally longer than the writing time of the non-professional users. The reason why the accuracy cannot reach 100% is that the requirements themselves are wrong, even though they satisfy consistency and completeness. For example, a professional user wrote the following requirement "IF *Room.humidity* $>$ 45 THEN *Window.open*". When the air humidity is greater than a certain value (during rain), the window is open, which deviates from the real life. Therefore, this is a wrong requirement that does not break the consistency and completeness. The professional users take longer on average because they have more background knowledge and think more about writing requirements, whereas the non-professional users rely entirely on the practical experience.

5.2 Performance

This section analyzes the performance of the automation process of the proposed approach. Five groups of experiments are designed for the performance evaluation of the algorithm of converting service requirements to problem diagrams, the consistency checking algorithm, the completeness checking algorithm, the TAP generation algorithm, as well as the whole approach in this paper. Ten experiments are designed for each group to run the algorithms on 10, 30, 50, 100, 200, 300, 500, 800, 1,000, and 1,200 requirements respectively, and the time cost is recorded. The experimental environment of this paper is 64-bit Win10 system, Intel(R) Core(TM) i7-9700k CPU @ 3.60 GHz, 32 GB RAM.

The service requirements for each experiment are automatically generated by the program. When the algorithm of converting the service requirements to the problem diagrams and the TAP generation algorithm are evaluated, the exactly correct service requirements are generated. When the consistency and completeness checking algorithms and the whole process are evaluated, the probabilities of inconsistent requirements and incomplete requirements in the generated requirements are both 20%. If the inconsistent requirements are generated, two kinds of inconsistencies are randomly generated with a probability of 1/2 respectively. If the incomplete requirements are generated, the state incompleteness, the attribute incompleteness without "`and`" and "`or`", and the attribute incompleteness with "`and`" and "`or`" are randomly generated with the probabilities of 1/4, 1/4, and 1/2, respectively. To avoid the influence of randomness on the evaluation results, we carry out each experiment 10 times. The number of inconsistency/incompleteness and the time cost of each experiment are recorded and the experimental results are averaged as the evaluation result. The average number of inconsistency/incompleteness of the ten groups of experiments is listed in Table 3, and the final results of performance evaluation are shown in Figure 8.

As can be seen from the change trend in Figure 8, the performance of the proposed approach exceeds the available threshold. The time costs of the algorithm of converting service requirements to problem diagrams, the TAP generation algorithm, and the consistency and completeness checking algorithms all increase with the number of requirements, but the increase rate is not very high, which is basically consistent with the algorithm complexity of O(n) and O(n^2) mentioned above. Therefore, this method can be applied to large-scale systems.

Figure 8 also shows that when the number of requirements increases to a certain value, the time cost of the algorithm of converting service requirements to problem diagrams is significantly longer than those of the consistency and completeness checking algorithms and the TAP generation algorithm. The reason is as below. Assuming that the number of requirements is n, the time complexity of the consistency and completeness hecking algorithms and the TAP generation algorithm is O(n), as described in Section 4. However, the complexity of the algorithm of converting service requirements to problem diagrams is O(n^2). Therefore, the time complexity of the converting algorithm is higher than the two checking algorithms and the TAP generation algorithm. With the increase in the number of requirements and thus the total number of states in the environment ontology, the converting algorithm is supposed to take more time.

Table 3 Average number of inconsistency/incompleteness and their distribution in experiments

Total number of requirements	Inconsistency		Incompleteness		
	Scope	Overwriting	State	Attribute without "and" and "or"	Attribute with "and" and "or"
10	0.4	0.8	0.3	0.3	0.8
30	0.8	2.6	1.9	1.0	2.2
50	3.8	2.9	2.8	2.9	4.0
100	7.3	7.6	3.7	4.8	10.0
200	10.8	12.6	9.7	10.8	19.6
300	19.5	21.1	14.0	13.9	31.5
500	32.4	30.6	25.2	28.1	45.7
800	50.9	54.1	40.0	39.1	80.9
1,000	63.2	64.8	49.0	51.9	91.5
1,200	81.3	77.7	61.2	58.5	120.7

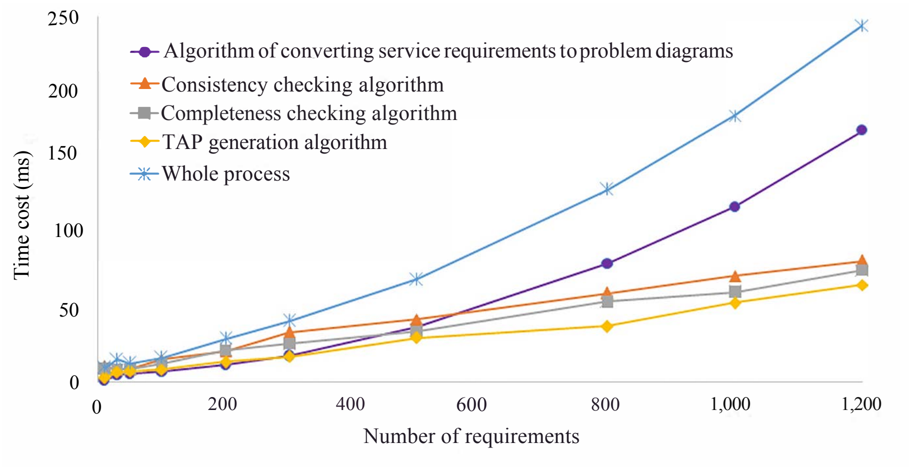


Figure 8 Result of performance analysis

In addition, the time cost of the consistency and completeness checking algorithms in Figure 8 is not strictly monotonically increasing. In our opinion, this is related to the number of inconsistency/incompleteness in the service requirements in each group of experiment, namely the time spent on checking consistent (complete) and inconsistent (incomplete) service requirements is different. To prove this point, we design another four groups of experiments for comparison. Groups 1 and 2 evaluate the performance of consistency and completeness checking algorithms when there is no consistency and completeness error respectively, and

Groups 3 and 4 evaluate the performance of consistency and completeness checking algorithms when there are consistency and completeness errors respectively. The average number of inconsistency/incompleteness in Groups 3 and 4 can still be seen in Table 3. Each group is divided into 10 experiments according to the number of service requirements, and each experiment is repeated 10 times, whose time cost is recorded. The results are averaged and the final results are shown in Figure 9.

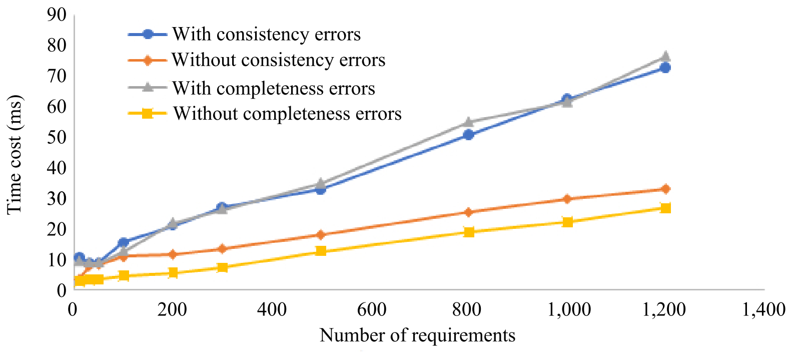


Figure 9 Time cost of consistency and completeness checking

When there are no consistency and completeness errors in the problem diagram, the time for consistency and completeness checking is greatly shortened (Figure 9). Therefore, when the proposed approach is used to generate TAP rules, if the requirements written by users are of high quality and have no errors, the time cost will not be increased due to the consistency and completeness checking.

5.3 Construction time of environment ontology

Compared with directly writing the TAP rules, the proposed approach needs to construct the environment ontology first. This section tries to discuss whether it is worth spending time to construct the environment ontology, namely whether the construction of the environment ontology can save more other time, and whether the time to construct the environment ontology can be ignored as the number of requirements increases.

In the accuracy and efficiency evaluation experiments in Section 5.1, the environment ontology contains 5 controlled entities, with a total of 11 states and 24 transition relationships. Although users spend 7 min on average to construct and modify the environment ontology, this method automatically checks the consistency and completeness errors and thus saves more time as compared with the manual checking. In addition, the proposed method also prompts the incorrect statements to help users make the modifications. As can be observed from Table 2, when there are 10 requirements, the time spent to construct the environment ontology has been less than the saved time. However, in the scenario with a large number of requirements, although the construction time of the environment ontology also increases to a certain extent due to the increased number of entities, it is inferior to the great time cost of the manual consistency and completeness checking. Therefore, we believe that it is worth spending some extra time to construct the environment ontology and thereby save more time.

This paper also designs a group of experiments to prove the above conclusions. We argue that although the number of requirements increases continually, if the total time spent on constructing the environment ontology, writing the requirements, deriving the system behaviors, and checking and modifying consistency/completeness until the generation of the rules tends to be stable, the impact of constructing the environment ontology on the efficiency can be ignored after the scale of requirements reaches a certain degree. Therefore, 10 experiments are designed and numbered 1 to 10. The ratio of total time (the sum of the time to construct the environment ontology and the time of the above steps) to the number of requirements, namely the average time required for each requirement, is calculated in the scenarios with 10, 30, 50, 100, 200, 300, 500, 800, 1,000, and 1,200 requirements respectively. The experimental results are shown in Figure 10.

When the number of requirements increases, the ratio of the total time to the number of requirements gradually tends to be stable (Figure 10), namely that the relative time to construct the environment ontology becomes shorter and even tends to 0. There are two reasons for this phenomenon. First, domain experts become more familiar with the construction of the environment ontology, and thus the construction of the environment ontology becomes faster. Second, many elements in the same domain can be reused. For example, for a smart home IoT system, when users write 10 requirements, the controlled entities involved in the environment ontology only contain bulb, window, and blind. When users write 50 requirements, the controlled entities involved in the environment ontology are bulb, blind, window, air conditioner, and TV. The environment ontology of bulb, window, and blind can be reused, which thereby reduces the construction time of the environment ontology. Under the influence of these two aspects, the time to construct the environment ontology is shared by more and more requirements and gradually becomes negligible.

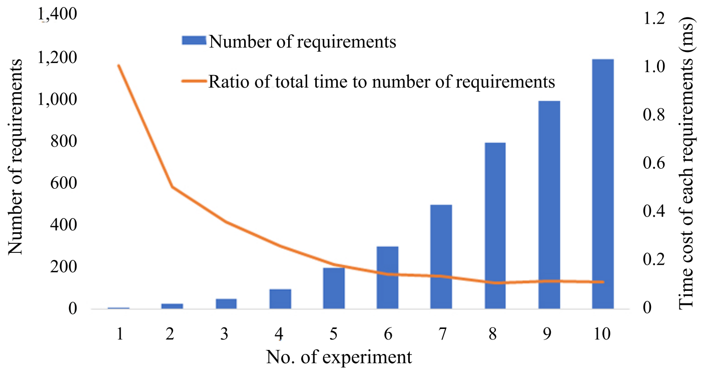


Figure 10 Ratio of total time to number of requirements

It is worth noting that the correct execution of the proposed approach depends on the correct environment ontology. If an error occurs in the environment ontology, the correctness of the method will be affected. If an environment entity that should not exist is mistakenly added, some requirements without problems will be wrongly judged as state incompleteness during the completeness checking. If there is a transition error in the environment ontology, such as an incorrect transition condition, the corresponding system behavior may not be found when the system behavior is deduced with the environment ontology, which thereby results in errors. In fact, this problem has already arisen in the user study in Section 5.1. A professional user spent 14 min modifying the requirements, which is quite a long time in the presence of mistake prompts by the approach. After inquiry, the user found a mistake when constructing the environment ontology in the process of modification, rebuilt the environment ontology and modified the requirements, so a long time was spent. Therefore, for the construction of the environment ontology, two points should be emphasized. The environment ontology should be constructed by domain experts; once constructed, the environment ontology should be peer reviewed for the minimization of the possible errors.

6 Related Work

The work related to this paper includes the requirement acquisition approach of the IoT system, the requirements consistency and completeness checking, and the consistency and completeness checking method of TAP. There are already some efforts to capture the requirements for the IoT systems. For example, Reference [2] introduces a set of methods for formulating and standardizing the requirements of the IoT systems. Requirements specification of the IoT systems includes domain model, target view describing the goals and their relationships, and goals' specification described by UML diagrams. It adopts the goal-oriented method concept[14] and denotes requirements as goals, which is relatively subjective. However, the requirements engineering concept based on environment modeling adopted by this paper denotes requirements as state changes of environment devices, which is relatively objective. In addition, the domain model in Reference [2] and the environment ontology in this paper all belong to the domain knowledge. However, it is only used for requirement acquisition, not for the consistency and completeness checking. Other work focuses on the non-functional requirements of IoT systems. For example, based on the $i*$ framework[16], Reference [15] captures the security and privacy requirements in the early stage of the development of IoT systems. On the basis of K-Model, Reference [17] allows developers to write the codes in JSON form according to the template and then write and modify the non-functional requirements through these codes. This paper only involves the functional requirements.

Much work has also been done on the requirements completeness and consistency checking. Reference [18] describes the disastrous consequences that the inconsistent and incomplete requirements bring to software development and proposes the indicators to measure the completeness and consistency of Software Requirement Specification (SRS), but the completeness and consistency need to be guaranteed manually. Reference [19] carries out the automatic consistency checking. The requirement is expressed as SCR tabular and the software system as finite state automata. The consistency is automatically checked by static analysis. Reference [20] uses a state-based requirement specification language, RSML, to express the requirements and makes use of the characteristics of RSML to analyze the completeness and consistency. To improve the requirements completeness, Reference [21] provides a natural language processing tool. During the writing of requirements, the tool automatically prompts the terms or the relationships that may be used to help the requirements engineers find relevant concepts and interactions, so that the requirements can be written more accurately. Depending on an obstacle analysis, Reference [22] combines the model checking with the machine learning to identify, evaluate, and solve abnormal situations that may hinder the system objectives, so as to produce complete requirements. However, they are not applied in the IoT systems and do not support user programming. In addition, they do not use the environment knowledge for checking.

There are also efforts to conduct the consistency and completeness checking based on knowledge. For example, Reference [8] proposes a knowledge-based requirements engineering process method to ensure the requirements consistency and completeness. The authors propose a hybrid model based on the framework ontology and the production rules to represent the system requirements. The combined ontology framework and production rules are used for the requirements consistency and completeness checking. The framework proposed in Reference [23] uses expert agents to assist users in requirement definition. It uses knowledge base and case base to help users define a set of system requirements that conform to the completeness and consistency. It also allows the control power to switch between users and expert agents, which provides a collaborative medium for requirements writing. Reference [24] presents an ontology-driven goal-based requirements engineering meta-model, which is based on reasoning technology and combined with the ontology consistency checking and the rule-driven completeness checking. This model can be used to both capture the requirements and measure the correctness and coverage of the requirement models. These knowledge-based methods are similar to this paper in some aspects. The environment ontology in this paper is also knowledge, but it is the environment knowledge about IoT systems.

The consistency checking of many requirements are the formalization and formal verification methods based on the pattern language. Reference [25] uses the restricted natural language in the form of the Specification Pattern System (SPS) to describe the system requirements and then automatically converts them into the timed computational tree logic and then into the first-order logic formulas. Z3 is used to perform the SMT analysis and check whether the requirements are consistent. Reference [26] adopts BTC mode to represent the requirements[27]. To minimize the analysis cost, it utilizes bounded model checking to detect consistency. Reference [28] defines a pattern language named SafeNL for safety requirements, which expresses the requirements in a quasi natural language and then automatically transforms it into the clock constraint specification language, and detects the consistency of security requirements by means of bounded model checking. Reference [29] proposes a new concept of formal consistency, namely the partial order consistency, to simplify the general pattern. Partial order consistency can identify the key cases of a system and verify whether these cases cause conflicts between the requirements. In this reference, the requirements expressed by the simplified general pattern are subjected to formal modeling by counting automata, and then the partial order consistency is formally defined. In addition, the method to detect whether the requirement model deviates from the partial order consistency is given. Reference [30] proposes a language for structured requirements of automobile systems. It uses the ontology of automobile systems to provide lexical and syntactic standards and reduces the requirements consistency checking into a proof of Boolean propositions. Afterward, it uses theorem proving to check the requirements consistency. Among these methods, model checking and theorem proving are both heavy-weight formal methods, whose efficiency is difficult to guarantee and returned results are difficult to be understood by ordinary users.

At present, formal methods are used to check the consistency of TAP rules. For example, AutoTap in Reference [7] allows users to utilize the TAP rules to describe the requirements and also to define the attributes that the system must satisfy. AutoTap transforms the rules and attributes written by the users into LTL formulas, and model checking is adopted to check whether they have conflicts or not. The tool developed by Zhang *et al*.[31] uses formal methods to transform the TAP rules into CTL and LTL formulas. Then their consistency is checked with the help of the NuSMV tool. Reference [32] also adopts a formal method to model the IoT system with the linear hybrid automata and then performs reachability analysis to check consistency. All these methods check the TAP rules and are quite different from this paper, in which whether there is a conflict between system behaviors is checked. Moreover, the approach in this paper is actually based on rules, with a high efficiency and a large scale of use.

7 Conclusions and Future Work

To realize the automatic generation from service requirements to device scheduling instructions and ask users to participate in the development of the IoT system, this paper proposes an approach to generating the TAP rules based on environment modeling. It automatically derives the system behaviors from the service requirements based on the environment model, checks the completeness and consistency of system behaviors, and finally generates the TAP rules. This paper constructs the environment ontology, provides the concepts and associations of environment modeling, and describes the state, attribute, behavior, and their relationship of each device in the IoT system. Then, depending on the environment ontology, this approach supports the whole process including the derivation of system behavior from the service requirements written by the users, the consistency/completeness checking, and TAP rule conversion. The evaluation results of the proposed method show that it has good performance, efficiency, and accuracy. The evaluation also indicates that as the number of requirements increases, the time to construct the environment ontology becomes negligible and saves plenty of time for manually checking consistency and completeness.

This paper only focuses on the functional requirements of IoT systems and does not involve non-functional requirements such as time, security, privacy, and reliability. Future work will be conducted to capture and check these non-functional requirements. In addition, the current construction of the environment ontology only relies on domain experts, and the next work will consider the semi-automatic or even automatic methods for construction.

References

[1] Klosowski T. Automation Showdown: IFTTT vs Zapier vs Microsoft Flow. LifeHacker, 2016.

[2] Reggio G. A UML-based proposal for IoT system requirements specification. Proc. of the 10th IEEE/ACM Int'l Workshop on Modelling in Software Engineering (MiSE). Gothenburg, Sweden. 2018. 9–16.

[3] Zowghi D, Gervasi V. On the interplay between consistency, completeness, and correctness in requirements evolution. Information and Software technology, 2003, 45(14): 993–1009.

[4] Doe J. Recommended practice for software requirements specifications. New York: IEEE, 2011. https://www.midori-global.com/ downloads/jpdf/jira-software-requirement-specification.pdf.

[5] IEEE Guide for Software Requirements Specifications. 1984. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=278253.

[6] Srivastava A, Patel F, Sivagami M.A software requirement engineering technique using OOADA-RE and CSC for IoT based healthcare applications. International Journal of Software Engineering & Application, 2018, 9(1): 55–63. [doi: 10.5121/ijsea.2018.9105]

[7] Zhang L, He W, Martinez J, *et al*. AutoTap: Synthesizing and repairing trigger-action programs using LTL properties. Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal, QC, Canada. 2019. 281–291.

[8] Avdeenko TV, Pustovalova NV. The ontology-based approach to support the requirements engineering process. Proc. of the Int'l Scientific-technical Conf. on Actual Problems of Electronics Instrument Engineering (APEIE). IEEE. 2016. 513–518.

[9] Chen X, Jin Z. Capturing requirements from expected interactions between software and its interactive environment: An ontology based approach. Int'l Journal of Software Engineering and Knowledge Engineering, 2016, 26(1): 15–39.

[10] Jin Z. Environment Modeling-based Requirements Engineering for Software Intensive Systems. Morgan Kaufmann Publishers, 2018.

[11] Jackson M. The meaning of requirements. Annals of Software Engineering, 1997, 3(1): 5–21.

[12] Jackson M. Problem frames: Analysing and Structuring Software Development Problems. Addison-Wesley, 2001.

[13] Studer R, Benjamins VR, Fensel D. Knowledge engineering, principles and methods. Data and Knowledge Engineering, 1998, 25(1/2): 161–197.

[14] Liu L, Yu E. Designing information systems in social context: A goal and scenario modeling approach. Information Systems, 2004, 29(2): 187–203.

[15] Alqassem I. Privacy and security requirements framework for the Internet of Things (IoT). Proc. of the Int'l Conf. on Software Engineering. 2014. 739–741.

[16] Yu E, Liu L. Modelling trust for system design using the *i** strategic actors framework. Proc. of the Trust in Cyber-societies. 2001. 175–194.

[17] Ferraris D, Fernandez-Gago C. TrUStAPIS: A trust requirements elicitation method for IoT. Int'l Journal of Information Security, 2020, 19(1): 111–127.

[18] Kuchta J. Completeness and consistency of the system requirement specification. Proc. of the Federated Conf. on Computer Science and Information Systems. 2016. 265–269.

[19] Heitmeyer CL, Jeffords RD, Labaw BG. Automated consistency checking of requirements specifications. ACM Trans. on Software Engineering and Methodology (TOSEM), 1996, 5(3): 231–261.

[20] Heimdahl MPE, Leveson NG. Completeness and consistency in hierarchical state-based requirements. IEEE Trans. on Software Engineering, 1996, 22(6): 363–377.

[21] Ferrari A, dell'Orletta F, Spagnolo GO, *et al*. Measuring and improving the completeness of natural language requirements. Proc. of the Int'l Working Conf. on Requirements Engineering: Foundation for Software Quality. 2014. 23–38.

[22] Alrajeh D, Kramer J, van Lamsweerde A, *et al*. Generating obstacle conditions for requirements completeness. Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). 2012. 705–715.

[23] Sinha AP, Popken D. Completeness and consistency checking of system requirements: An expert agent approach. Expert Systems with Applications, 1996, 11(3): 263–276.

[24] Siegemund K, Thomas EJ, Zhao Y, *et al*. Towards ontology-driven requirements engineering. Proc. of the Workshop Semantic Web Enabled Software Engineering at the 10th Int'l Semantic Web Conf. (ISWC). Bonn, Germany. 2011.

[25] Filipovikj P, Rodriguez-Navas G, Nyberg M, *et al*. SMT-based consistency analysis of industrial systems requirements. Proc. of the Symp. on Applied Computing. 2017. 1272–1279.

[26] Ellen C, Sieverding S, Hungar H. Detecting consistencies and inconsistencies of pattern-based functional requirements. Proc. of the 19th Int'l Conf. on Formal Methods for Industrial Critical Systems (FMICS2014). 2014. 155–169.

[27] BTC Embedded Systems AG: BTC Embedded Validator Pattern Library, Release 3.6(2012).

[28] Chen X, Zhong Z, Jin Z, *et al*. Automating consistency verification of safety requirements for railway interlocking systems. Proc. of the 27th IEEE Int'l Requirements Engineering Conf. (RE). 2019. 308–318.

[29] Becker JS. Analyzing consistency of formal requirements. Proc. of the Electronic Communications of the EASST. 2019. 76.

[30] Mahmud N, Seceleanu C, Ljungkrantz O. ReSA: An ontologybased requirement specification language tailored to automotive systems. Proc. of the 10th IEEE Int'l Symp. on Industrial Embedded Systems (SIES2015). 2015. 1–10.

[31] Zhang QP, Wang XZ, Shen SY, *et al*. Automated configuration, simulation and verification platform for event-driven home automation IoT system. Wu Lian Wang Xue Bao/Chinese Journal on Internet of Things, 2019, (3): 90–101.

[32] Lei B, Wen X, Mike CJ, *et al*. Systematically ensuring the confidence of real-time home automation IoT systems. ACM Trans. on Cyber Physical Systems, 2018, 2(3): 1–23.

Han Bian bachelor, CCF student member. His research interests include IoT and requirements engineering.

Zhi Jin Ph.D., professor, doctoral supervisor, CCF fellow. Her research interests include requirements engineering and knowledge engineering.

Xiaohong Chen Ph.D., associate professor, CCF professional member. Her research interests include requirements engineering, formal methods, and IoT systems.

Min Zhang Ph.D., professor, CCF professional member. His research interests include trustworthy software theory and formal methods.