PAPER Special Section on Knowledge-Based Software Engineering

Improvement of the Correctness of Scenarios with Rules

Atsushi OHNISHI[†a)], ***Member***

SUMMARY Scenarios that describe concrete situations of software operation play an important role in software development, especially in requirements engineering. Since scenarios are informal, the correctness of scenarios is hard to be verified. The authors have developed a language for describing scenarios in which simple action traces are embellished. The purposes are to include typed frames based on a simple case grammar of actions and to describe the sequence among events. Based on this scenario language, this paper describes both (1) a correctness-checking method using rules to detect errors (lack of events, extra events, and wrong sequence among events) in a scenario and (2) a retrieval method of rules from rule DB that applicable to scenarios using pre and post- conditions.

key words: *scenario analysis, requirements elicitation, requirements definition, rule-based checking*

1. Introduction

Scenarios are important in software development [5], particularly in requirements engineering, since they provide concrete system description [20], [22]. Moreover scenarios are useful in defining system behaviors done by system developers and validating the requirements undertaken altogether with customers. In many cases, scenarios become foundation of system development. Incorrect scenarios will lead to negative impact on system development process in overall. However, scenarios are informal and it is difficult to improve the correctness of scenarios. The errors in incorrect scenarios may include:

1. Vague representations,
2. Lack of necessary events,
3. Extra events,
4. Wrong sequence among events.

The author has developed a scenario language for describing scenarios in which simple action traces are embellished to include typed frames based on a simple case grammar of actions and to describe the sequence among events [14], [15], [23]. Since this language is a controlled language, the vagueness of the scenario written using this language can be reduced [13], [14]. Furthermore, the scenario created with this language can be transformed into internal representation. In the transformation, both lack of cases and illegal usage of noun types can be detected, and concrete words will be assigned to pronouns and omitted indispensable cases [13], [15]. As a result, the scenario with this language can avoid errors type 1 previously mentioned.

This paper focuses on how to detect the errors typed 2, 3, or 4 in scenarios. The authors propose a method of using rules to detect the errors (lack of events, extra events, and wrong sequence among events) in scenarios.

To lessen the job of making rules, the authors also provide a rule DB and an automatic selection method of rules that are applicable to scenarios using pre-conditions and post-conditions specified in both rules and scenarios.

The rest of this paper is organized into 6 sections. Section 2 introduces the outline of the scenario description language and gives a scenario example. Section 3 describes in detail the scenario checking method with rules. Section 4 illustrates the method with an example. Section 5 describes evaluation of our method through an experiment. In Sect. 6, a discussion of related works is presented. Finally, in Sect. 7 we provide some concluding remarks and point out our future works.

2. Scenario Language

Our scenario language has been already introduced in several papers such as in [14], [15], [23]. However, in this paper, a brief description of this language will be given for convenience.

A scenario can be regarded as a sequence of events. Events are behaviors employed by users or systems for accomplishing their goals. We assume that each event has just one verb, and that each verb has its own case structure. The scenario language has been developed based on this concept. Verbs and their own case structures depend on problem domains such as elevator control [14], PC chair's job [3], [23] and train ticket reservation [16], but the roles of cases are independent of problem domains. The roles include agent, object, recipient, instrument, and source, etc. [6], [15].

We provide Requirements Frames [13] in which verbs and their own case structures are specified. This frame depends on problem domains. Each action has its case structure, and each event can be transformed into internal representation based on this frame. In the transformation, concrete words will be assigned to pronouns and omitted indispensable cases. With Requirements Frame, we can detect both the lack of cases and the illegal usage of noun types [13].

We assume four kinds of time sequences among events: 1) sequence, 2) selection, 3) iteration, and 4) parallelism. Actually most events are sequential events. Our scenario

Manuscript received June 27, 2005.
Manuscript revised October 3, 2005.

[†]The author is with Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
a) E-mail: ohnishi@cs.ritsumei.ac.jp

DOI: 10.1093/ietisy/e89–d.4.1337

[Scenario title: A customer purchases a train ticket of reservation seat]
[Viewpoints: Staff, customer]
[Pre-condition: the customer has enough money to buy a ticket & has not a ticket & has not reserved a seat]
[Post-condition: the customer will get a ticket & reserved a seat]
1. A staff asks a customer leaving station and destination as customer's request.
2. He sends the customer's request to reservation center with a terminal.
3. He retrieves available trains with the request.
4. He informs the customer of a list of available trains.
5. The customer selects a train that he/she will get.
6. The staff retrieves available seats of the train.
7. He shows a list of available seats of the train.
8. The customer selects a seat of the train.
9. **If** *(there exists a seat selected by the customer)* **then** *the staff reserves the seat with the terminal.*
10. He gets a permission to issue a ticket of the seat from the center.
11. He receives money for the ticket from the customer.
12. He gives the ticket to the customer

Fig. 1 Scenario example.

language defines the semantic of verbs with their case structure. For example, data flow verb has source, goal, agent, and instrument cases. Since such case structure can define the abstraction level, scenario provided using our scenario language becomes the almost same level of the abstraction.

We consider a scenario of train ticket reservation in a railway company. Figure 1 shows a scenario of customer's purchasing a ticket of express train at a service center. This scenario is written with our scenario language based on a video that records behaviors of both a user and a staff at one particular service center.

A title of this scenario is given at the first two lines in Fig. 1. Viewpoints of considered scenario are specified at the third line. In this paper, viewpoints mean active objects such as human or system appearing in the scenario. There exist two viewpoints, namely staff and customer. The order of specified viewpoints means the priority. In this example, the first featured object is staff and the second one is customer. In such a case, the former becomes the subject of an event.

In addition, pre-condition specifies a condition that satisfies at the start of the scenario. Post-condition specifies a condition that satisfies at the end of the scenario.

In this scenario, most events are sequential, except one selective event (the 9th event). Selection can be expressed with if-then syntax like program languages. Actually, event number is for reader's convenience and not necessary.

3. Correctness-Check of Scenarios

When a scenario is described, necessary events may be missing, unnecessary events may be mixed or time sequence among events may be inaccurate. These errors may have a negative impact on system development; therefore, it is necessary to detect these errors. The errors, also employed as correctness checked items, of scenarios include:

1. Lack of necessary events
2. Extra events
3. Wrong time sequence among events

We can check whether an event is lacking, being extra one or being sufficient one by comparing its correct occurrence times with the times that it occurred in the scenario. Similarly, we can check whether the time sequence among events is wrong by comparing the correct time sequence with the time sequence described in the scenario.

We propose a method to improve the correctness of scenarios by using rules to detect the errors in scenarios. We assume that a rule is a description of the correct occurrence times of an event and/or the correct time sequence among events, which the scenario ought to satisfy. One scenario may be checked with several rules.

3.1 Rule

Rule is composed of the description of rule's event and the description of event's occurrence times and/or time sequence among events. In this sense, our rules just specify the occurrence of events and/or the sequence of events. If the abstraction level of events of rules becomes high, the rules can be applied to scenarios of several different domains.

3.1.1 Events in Rule

In a rule, there are one or more events, whose occurrence times and/or time sequence are specified. When a scenario is checked with a rule, the rule's events can correspond to the scenario's events. By finding the corresponding events in the scenario, and by checking the occurrence times and/or the time sequence of these events, one scenario can be checked.

As a result, it is necessary to get the corresponding relation between the rule's events and the scenario's events. For this reason, the rule's events are also described based on Requirements Frames, and can also be transformed into the internal representation. If the rule's event and the scenario's event have the same internal representation, then they are deemed as corresponding events.

In order to improve the correctness-checking effect, it is not sufficient that the corresponding relation between the rule's event and the scenario's event is 1 to 1 ratio. It is expected that a rule's event can correspond to several scenario's events. As a result, the occurrence times and/or the time sequence of these scenario's events can be checked with one rule. In such a case, the rule's event has an abstract representation, and the corresponding scenario's events have several concrete representations.

For the above reason, we permit the abstract description of rule's event. An abstract event may be transformed into several concrete events, when finding its corresponding

events in the scenario. At this time, the corresponding relation between the rule's event and the scenario's event is 1 to many. There are two kinds of abstract events in the rule.

1. Some indispensable cases are omitted in the event sentence.
2. There include "something," "someone," "same thing," "same one," etc. in the event sentence.

In the first kind of abstract events, the omitted cases fit any noun. This kind of abstract events will be transformed into concrete events by substituting concrete nouns for omitted cases, when finding its corresponding events in observed scenario. For example, a rule's event "system feedbacks to user" can be transformed into an internal representation.

This event can correspond to any scenario's event whose action is "feedback, etc.", agent case is "system", and recipient case is "user". In the second kind of abstract events, "something" / "someone" is similar to the omitted case in the first kind of abstract events, and fit anything / anyone. The reason of dividing abstract events into two kinds should be explained. Assuming that there is a rule that describes the time sequence among events. Under this rule there exist two separate events and under these two events there exist a case A and a case B. We want to specify that case A and case B can have any content but they have to be the same content. If we simply omit case A and case B in the rule description, it cannot be warranted that case A and case B have the same content. By specifying "something" / "someone" for case A, "same thing" / "same one" for case B, case A and case B can have any content, and they are the same content.

In the second kind of abstract events, "same thing" / "same one" fits the same noun with "something" / "someone" that appears in the same rule.

This kind of abstract events will be transformed into concrete events by substituting concrete nouns for "something" / "someone" / "same thing" / "same one", etc. when finding its corresponding events in the scenario.

3.1.2 Occurrence Times of an Event

The correct occurrence times of an event, which the scenario ought to satisfy, can be specified as a rule. By comparing the correct occurrence times with the times that this event occurred in the scenario, whether this event is lack of or excess of occurrence can be checked. The occurrence times of an event are described based on regular expression as follows.

1. E: event E occurs just one time.
2. $E+$: event E occurs one or more times.
3. $E?$: event E occurs one time or does not occur.
4. $E!$: event E never occurs.
5. $E\{m\}$: event E occurs m times.
6. $E\{m,\}$: event E occurs m or more times.
7. $E\{m,n\}$: the occurrence times of event E is from m to n.

We adopted the syntax of regular expression and similar its semantics to specify the occurrence time of events in rules.

3.1.3 Time Sequence among Events

The correct time sequence among events that scenario ought to satisfy can be specified as rules. By comparing the correct time sequence with the time sequence described in the scenario, time sequence among events can be checked. According to the time sequence in the scenario described in Sect. 2.1, we assume the following rules.

1. *Before/After E1, E2*: Before/After event E1 occurs, event E2 should occur.
2. *If* (*condition*) (*E1, E2*): Event E1 and event E2 occur selectively. If the condition is true, event E1 occurs. If the condition is false, event E2 occurs.
3. *Do* (*E1, E2*, ...) *until* (*condition*): Until the condition becomes true, event E1, E2, ... occur iteratively.
4. *AND* (*E1, E2*, ...): All of the events E1, E2, and others parallel occur.
5. *OR* (*E1, E2*, ...): One of the events E1, E2, ... or more parallel occurs.
6. *XOR* (*E1, E2*, ...): Just one of the events E1, E2, ... occurs.

As previously described, when a scenario is checked with a rule that includes the abstract event, it is possible that an abstract event corresponds to several scenarios' events. In this case, it is necessary to check the time sequence of every corresponding event in the scenario and the results should be shown one by one.

3.1.4 Classification of Rules

According to the applicable extent, rules can be classified into three categories.

1. Scenario-specific rules
2. Domain-specific rules
3. Domain-independent and generic rules

A scenario-specific rule is one, which could be true for particular scenarios and false for the others. For example, a rule, "*Before a customer selects a seat, he/she selects an available train in advance*" is an example of scenario-specific rules. This rule can be applicable to some scenarios of train ticket reservation systems, but cannot be applicable to most scenarios belonging to the rental domain. We think that train ticket reservation is not a domain but belongs to the rental domain. This kind of rule specifies the sequence of between concrete events or the occurrence of concrete events.

On contrast, domain-specific rules describe more abstract and more fundamental features of one problem domain, and therefore can be available for the scenarios belonging to these problem domains. A rule "*Before customer receives a ticket, reservation center permits for issuing a ticket*" is an example of domain-specific rule. This rule becomes true for most scenarios of the rental domain including ticket reservation systems.

Domain-independent and generic rules describe quite

```
Initialize a counter (counter=0).
Find the scenario's event that corresponds to the rule's event
while (the corresponding event in the scenario will be found)
  do
    counter=counter+1
    Show the corresponding event and its occurrence condition to
      user.
    Find the next scenario's event that corresponds to the rule's
      event.
  od
Compare the occurrence times specified in the rule with the
counter, and show the result.
```

Fig. 2 Checking procedure of the occurrence of events.

fundamental features of most problem domains. This type of the rule becomes true for most scenarios that are independent of domains. A rule "*Before sending something, the same thing should be* [*made*|*received*] *in advance*" is an example of domain-independent and generic rule.

Domain-specific rules and domain-independent and generic rules can be prepared beforehand based on the fundamental features of problem domains [8], [10] and stored into a rule DB. Users can check the scenarios with these rules and do not need to write them. Contrarily, scenario-specific rules are written by users and are not stored into the rule DB. A scenario can be checked with existing rules or ones provided by rule DB and/or newly specified rules.

Since there exist a lot of rules, valid rules should be selected and applied to a given scenario for its checking. A domain-specific rule becomes true for a normal scenario but may be false for alternative scenarios. We propose a method to automatically select valid rules for a scenario from a rule DB using both pre-conditions and post-conditions.

3.2 Scenario-Checking with Rule

Our scenario-checking procedure consists of two phases. The first phase is selection of applicable rules from rule DB. We specify both (1) pre-conditions and post-conditions and (2) viewpoints in each of the rules. When the conditions and viewpoints of a rule are much the same as those of a given scenario, the rule is selected for checking the scenario.

The second phase is analysis of rules and checking the consistency between a rule and the scenario as described in Sect. 3.1. The result will be passed to a checking-system user. The scenario checking can be achieved by automatically checking whether the scenario satisfies the rules, through the internal representation of scenario and the internal representations of rules.

We firstly find events in a scenario each of which corresponds to an event in a rule as described in Sect. 3.1.1. When a scenario is checked with a rule, the occurrence times and/or the time sequence of corresponding events in the scenario will be checked. We provide two checking procedures. One is for checking the occurrence time and the other is for checking the time sequence. Figure 2 shows outline of checking procedure for the occurrence time of an event and

```
Find the scenario's event that corresponds to E1 from the
beginning of the scenario.
if (the corresponding event of E1 not be found)
 then show this error, and the checking ends.
 else do
      Show the corresponding event and its occurrence condition.
      Find an event that corresponds to E2 and satisfies the time
      sequence
       if (the corresponding event of E2 not be found)
        then show the result that the scenario does not satisfy the
          rule
        else do
            Show the corresponding event and its occurrence
              condition.
            Find the next scenario's event that corresponds to E2
              and satisfies the time sequence.
           until (the corresponding event of E2 not be found)
       fi
      Find the next scenario's event that corresponds to E1.
     until (the corresponding event of E1 not be found)
fi
```

Fig. 3 Checking procedure of the sequence of events.

Fig. 3 shows outline of checking procedure for the time sequence between events E1 and E2. We have developed a prototype system with Java based on our method.

4. Example of Scenario Improvement

4.1 Checking with Domain-Independent Rules

As an example, we use domain-independent rule "*Before sending something, the same thing should be* [*made*|*received*]" to check the sample scenario in Fig. 1. This rule has no pre-conditions and post-conditions. This fact means that this rule can be applied to any scenarios. This rule includes two abstract events. One is "sending something" and the other is "the same thing should be [made|received]". The former event has "something." When this event leads to a corresponding event in the scenario, "something" can fit any noun of data type. The latter has "same thing", and when finding the corresponding event in the scenario, "same thing" should fit the same noun with the "something" of the former. Furthermore, these two events include omitted cases and when finding the corresponding event in the scenario, the omitted cases can match any noun.

First, we find the corresponding event of "sending something" in the scenario displayed in Fig. 1. Since the verb "send" is transformed into the conceptual verb, Data Flow [23], not only verb "send" but also "inform," "receive," and "give" are matched. As for the second event, "He sends the customer's information to reservation center with a terminal," sent data is "customer's information." In accordance with this rule, "customer's information" should be made or received in advance. In this case, since he has already received customer's information at the previous event, the rule is consistent with the scenario. As for the 4th event, a list of available trains should be made or received by the staff in

advance. In this case, the information is missing. The 3rd event should be "He retrieves available trains and get a list of available trains." The 11th event means that the customer has money and sends it to the staff. Since the customer has enough money as specified in the pre-condition, this event is consistent with the rule. As for the 12th event, the staff should have already issued the ticket before the 12th event. So there is a lack of event of issuing the ticket in this scenario. Using the domain-independent rule, we can detect two events are missing in the scenario of Fig. 1.

4.2 Checking with Domain-Specific Rules

In case of domain-specific rules, we specify pre-conditions and post-condition with each rule in advance. These two conditions are specified in scenarios and if both the pre-conditions and the post-conditions are same between a rule and a scenario, we regard that this rule can be applied to the scenario.

We provide a list of pre-conditions and post-conditions beforehand. It seems difficult to provide such conditions for a given system in advance, however, we assume that we can get them for a given domain through a neat domain analysis. Rule definers and scenario describers can select conditions from the list and specify them. Figure 4 shows a part of condition list. In this figure there are 8 conditions, where C2, C4, C6, and C8 are negations of C1, C3, C5, and C7, respectively. We provide 8 conditions for per-conditions and 10 conditions for post-conditions in a rule DB.

Table 1 shows domain-specific rules for ticket reservation. The rental item in the conditions C5 and C6 differs in case of the specific rental systems. In the train ticket and theater ticket reservation, the rental item will be seats. In the car rental system, the item will be a car.

C1: the customer has enough money to buy a ticket.
C2: the customer has not enough money to buy a ticket.
C3: the customer has a ticket.
C4: the customer does not have a ticket.
C5: the customer has already reserved a rental item.
C6: the customer has not yet reserved a rental item.
C7: the service is closed.
C8: the service is available.

Fig. 4 A part of the list of pre-/post-conditions for ticket reservation.

Table 1 A part of domain-specific rules for ticket reservation of rental domain.

Viewpoint	**Precondition**	**Postcondition**	**Rule**
Staff, customer	C3 &C6&C8	C3 &C5	After receiving customer's changing request, then staff retrieves DB
Staff,	C1&C4& C6&C8	C3&C5	After issuing a ticket, staff passes the ticket
Staff	C1&C4& C6&C8	C3&C5	After receiving money, staff passes the ticket.

As for the first rule in Table 1, the pre-conditions are "C3 & C6 & C8." This means C1/C2 are don't care conditions. To apply domain-specific rules to a certain scenario, we check the pre and post-conditions. Using the scenario in Fig. 1, the 2nd and the 3rd rules can be applied to this scenario, because both (1) the pre-conditions between the rules and the scenario and (2) the post-conditions between the rules and the scenario exactly match respectively. Since the 2nd rule is inconsistent with the scenario, we can detect that an event of issuing a ticket is missing. The 3rd rule is consistent with the scenario.

The domain-independent rules and domain-specific rules are provided as a rule DB in advance. So scenario describers do not need to define such rules, but it is difficult for them to select proper rules for check of a given scenario. We solve this problem by using pre and post-conditions and provide a method of automatically selecting rules for checking.

4.3 Checking with Scenario-Specific Rules

Scenario describers can write scenario-specific rules. Since these rules specify particular features of some specific scenarios, we do not provide them as a rule DB.

5. Evaluation

5.1 Experiment 1: Scenario of Train Ticket Reservation

In order to evaluate our method, the following experiment was performed. The purposes of the experiment are to confirm the following three points.

(1) Users can select proper pre and post-conditions for check,
(2) Our method can select proper rules using selected conditions, and
(3) Our method can contribute to detect errors in a scenario.

The experiment was performed in "Advanced Software Engineering" lecture of one graduate school. There are 15 graduate students majoring in computer science in the class. They were aware of the knowledge and experience about scenario, and 90 minutes were given to study our scenario description language and do some exercises. The train ticket reservation via WWW [16] was adopted as a problem for the experiment.

A prepared problem description written in natural language was provided to the students, and then 30 minutes were allocated to explain and comprehend the problem description, and 60 minutes were taken to describe scenario with the scenario language based on provided problem description. As a result, we got 15 scenarios for the problem.

This problem belongs to the rental domain. In this

Table 2 Specified pre and post-conditions in scenarios.

	Total	Correct	Wrong
Pre-conditions	48	27	21+18
Post-conditions	36	21	15+9

domain, customers can make reservations by specifying the reservation requests. After the reservation system received customers' reservation requests via Internet, the system searches whether available resources can be reserved or not. Subsequently, the system informs the customer about the search results, goods or services that are suited to the reservation requests. The customer selects some goods or services from the search results. Then the system searches databases for these goods or services, checks their availability again, informs the customer, and finally makes the reservation.

Prior to the experiment, domain specific rules with pre and post-conditions were provided to check the scenarios. Both the pre-conditions and the post-conditions in scenarios are same as those of rules were selected and applied. Through the experiment, we found that it is difficult for scenario describers to specify pre and post-conditions correctly. Table 2 shows that the ratio that describers could specify conditions correctly or not.

Since correct scenario has three pre-conditions and two post conditions, there should be 45 pre-conditions and 30 post conditions for 15 scenarios. There existed 48 pre-conditions and 36 post-conditions in 15 scenarios. We checked them and found that 27 pre-conditions were correct, but 21 pre-conditions were wrongly specified and 18 pre-conditions were not specified. We also found that 21 post-conditions were correct, but 15 post-conditions were wrong and 9 post-conditions were not specified.

We also found that scenario writers do not feel freely to describe scenarios if they should select pre and post-conditions from the list of conditions. If the describers can write these conditions informally, it becomes difficult to analysis the conditions by computers.

In order to solve these problems, we provided nine yes-no questions to identify pre and post-conditions, such as "At the beginning of the scenario, does the customer have any ticket. (Y/N)", "At the beginning, has he already reserved a train? (Y/N)", and "As a result, does the customer get a ticket as he intended? (Y/N)" and so on. This idea of using yes-no questions is quite effective to easily and correctly select rules without pre and post-conditions and does not prevent the flexibility of scenarios.

Table 3 shows customized domain-specific rules shown in Table 1. In this system, customers can directly buy tickets without any staffs, so staff in Table 1 corresponds to system.

In this experiment, since the pre-conditions of the scenario were C1 & C4 & C6 and the post-conditions of the scenario were C3 & C5, the 2nd and the 3rd rules were applied for the check. Here C1, C2, C3, C7 are same as the conditions in Fig. 4.

Table 4 shows the number of rules both for selected by scenario writes and selected automatically based on our method. Since the number of appropriate rules that are applicable to the scenario is nine, there are 135 rules for 15 scenarios. Scenario writers selected directly from the set of the rules by hand 120 rules. We checked them and found that 78 rules were correct, but 42 rules were wrong and 57 rules were not selected, while all of the selected rules by our method were perfect.

Table 3 Customized domain-specific rules.

View-point	Pre-condi-tions	Post-condi-tions	Rule
System, customer	C6	C5	After receiving customer's changing request, then system retrieves DB
System	C1&C4 &C6	C3& C5	After issuing a ticket, system passes the ticket
System	C1&C4 &C6	C3& C5	After receiving money, system passes the ticket

Table 4 Selected rules.

	Total	Correct	Wrong
Selected by hand	120	78	42+57
Automatically selected	135	135	0

Table 5 Detected errors in scenario.

	The number of errors	The number of detected errors
Wrong sequence of events	16	14
Lack of events	8	6
Extra events	11	2
Total	35	22

Actually there exist 21 rules in the rule DB and nine rules are selected with the matching of pre and post-conditions between rules and scenarios. These nine rules are exactly applicable to scenarios. The number of errors detected by checking the 15 scenarios with these rules was counted. Furthermore, the experiment manager prepared a correct scenario for each problem based on the problem description. By comparing the 15 obtained scenarios with the correct scenario, the experiment manager counted the number of all errors in the 15 obtained scenarios.

There exist 35 errors in 15 scenarios. These 35 errors can be classified into three categories, namely (1) wrong sequence of events, (2) lack of events, and (3) extra events. The number of errors grouped into the above categories are 16, 8 and 11, respectively. We could detect part of these errors with our method shown in Table 5.

The detection ratio is 63%. The reason why our method seems to be weak for detecting extra events is that it is very difficult to predict extra events and make rules for them in advance. On contrast, it is not so difficult to predict indispensable events and correct sequence of events and make rules for them in advance. Another reason why our method

is strong for detecting wrong sequence of events and lack of events is that rules for the occurrence time of events are effective to detect lack of events and rules for the sequence of events are effective to detect wrong sequence of events. To improve the detection ratio, we have to introduce another type of rules for detecting extra events.

The describers can easily correct the detected errors. Since scenario writers can determine the abstraction level of scenarios, the number of events may differ depending the scenario writers. Causes of undetected errors related to both lack of events and extra events are misunderstandings of the reservation jobs.

5.2 Experiment 2: Scenario of PC Chair's Job

We experimented for another problem domain. The problem is chair of Program Committee's job [3]. The experiment was performed in the lecture of the experiment 1. At that time there were 24 students in the class. 30 minutes were given to understand the problem domain and job description. In this experiment, we applied already prepared rules to scenarios developed by graduate students. The purpose of this experiment is the clarification of the following two items.

(1) How many errors in scenarios can be detected with domain-independent/domain-specific rules?
(2) How many errors in scenarios can be detected with scenario-specific rules?

As the domain-independent rules, we provide one rule in advance listed below.

I. Rule (di1): Before sending something, the same thing should have been made or received.

As the domain-specific rules, we provide two rules in advance.

II. Rule (ds1): After selecting some person, ask to the same person.
III. Rule (ds2): After being asked by someone, reply to the same person.

As the scenario-specific rules, we provide three rules in advance.

IV. List of submitted papers are sent via e-mail or FTP
V. Papers will be assigned to reviewers after reviewers sending keywords of research fields
VI. Papers are reviewed until all of the assigned papers will be reviewed

There are 56 errors in 24 scenarios. 18 errors can be detected with domain independent/specific rules and 48 errors can be detected with all of the rules. Table 6 shows the classification of errors and the number of detected errors. The detection ratio is 86% (48/56).

5.3 Experiment 3: Scenario of Car Insurance Problem

The third experiment was performed as similar way of the second experiment. We provide one domain-independent rule and one domain-specific rule and four scenario-specific rules.

Table 6 Detected errors in scenario (1).

	The number of errors	The number of detected errors	
		Domain-specific/independent rules	Scenario-specific rules
Wrong sequence of events	22	5	11
Lack of events	28	13	13
Extra events	6	0	6
Total	56	18	30

Table 7 Detected errors in scenario (2).

	The number of errors	The number of detected errors	
		Domain-specific/independent rules	Scenario-specific rules
Wrong sequence of events	19	0	17
Lack of events	90	67	23
Extra events	5	0	2
Total	114	67	42

As the domain-independent rules, we provide one rule in advance.

VII. Rule (di1): Before sending something, the same thing should have been made or received.

As the domain-specific rules, we provide one rule in advance.

VIII. After some document is presented, the same one will be checked.

As the scenario-specific rules, we provide four rules.

IX. If the presented document is insufficient to be checked, the document will be revised until the document becomes sufficient.
X. After refusing the request, notify to the requester
XI. If the insurance is valid, then negotiate the fee else refuse the request
XII. After the fee is accepted, compensation will be paid

There are 114 errors in 24 scenarios. 67 errors can be detected with domain independent/specific rules and 109 errors can be detected with all of the rules. Table 7 shows the category of errors and the number of detected errors. The detection ratio is 96% (109/114).

Through the experiments, we found the rule-based checking is effective to detect errors in scenarios. Not only domain independent/specific rules but also scenario-specific rules enable to detect most errors in scenarios.

6. Related Works

The team Cooperative Requirements Engineering With Scenarios (CREWS) of European Strategic Program for Research in Information Technology (ESPRIT) project conducted a survey through site visits with fifteen projects in four European countries for scenario-based requirements engineering [22]. The result was that the variety of purposes and usage of scenarios in the process was much greater than expected. As a result, we must take scenarios much more seriously as important design artifacts by offering better means for their description, management, and evolution. In order to handle these complex processes, users request more explicit methodological guidance and more adequate tool support.

In recent research papers, there is an obvious trend to define scenarios as textual description of the designed system behaviors. The growing number of practitioners demanding for more "informality" in the requirements engineering process seems to confirm this trend. Most of these papers describe how to use scenarios for the elicitation [19] or exploration [9] of requirements. The authors believe it is also important to provide help in the process of scenario description and define supports for how scenarios could be described with high quality in requirements engineering. The survey conducted by CREWS [22] also arrived at the similar conclusion. In this paper, the authors are interested in defining supports for semi-formal scenario description, and provide the scenario description language and verification method.

In the authors' previous work [13], we proposed to build software requirements in VRDL (Visual Requirements Definition Language) [13] from textual requirements in Japanese based on a typology of concepts that is very similar to the semantic roles of the case grammar [6]. Another related work is Achour's use of case grammar in scenario analysis [1], [2], [17]. Achour focuses on how textual scenarios could be integrated into different existing methods, and proposes guidance for writing scenarios. He provides style and content guidelines referring to conceptual and linguistic model of scenarios, based on the case grammar. These works demonstrate that the case grammar is suitable to the semantic characterization of any design models as well as the semantic characterization of any natural language sentence.

Achour proposes guidance for correcting scenarios based on a set of rules [1], [2]. These rules aim at the clarification, completion and conceptualization of scenarios, and help the scenario author to improve the scenarios until an acceptable level in terms of the scenario models proposed in the guidance is reached. Achour's rules can only check whether the scenarios are well written according to the scenario models. Nonetheless, it is possible that there are some missing events and incorrect time sequence of events in scenarios. We propose a verification method that can check both the semantics and the syntax of scenarios [21]. Applying this method, missing events and incorrect time sequence of events can be detected.

Sutcliffe et al. propose a verification method of scenarios based on validation-frame [20]. This frame consists of situation part and requirements part. In situation part, pattern of events and actions are defined. In requirements one, some generic requirements are needed to handle each of these patterns. Using validation-frame, crosschecking between scenario and requirements is possible. Our approach is similar, but we enable to check (1) wrong sequence of events and (2) the number of occurrence of events. On contrast, validation-frame does not check them.

7. Conclusion

The author proposed a scenario checking method with rules. We can specify the occurrence times of events and/or the time sequence among events as rules. Both scenario and rules can be transformed into the internal representation so that we can check scenario with rules and evaluate the correctness of one particular observed scenario.

The proposed method was demonstrated by the example and was evaluated. The evaluation results show that errors (the lack of events, extra events, and the wrong sequence among events) in scenario can be effectively detected by checking the scenario with rules. By using this correctness checking method, we can get scenarios, which can be used, more effectively in system development.

We also introduced a method of selecting rules applicable to a given scenario with pre-conditions and post-conditions. Automatic selections of applicable rules contribute to improve (1) the reusability of rules and (2) the correctness of applicable rules.

In this paper, we provide a list of pre-conditions and post-conditions in advance and also provide yes-no questions to specify pre and post-conditions of scenarios. Actually, for specific job belonging to a certain domain, it may be easy to provide such a list.

The ratio of detected errors is 63% in our experiment 1. Most (83%) errors of wrong sequence of events and lack of events can be detected, but errors of extra events cannot be well detected. Most undetected errors can be detected by enhancing rules in the DB, even so the enhancement of rule DB may lead less quality of the DB, since most of rules to be enhanced are not so generic ones. The detection ratio can be improved by using scenario-specific rules. The ratio is 86% at the experiment 2 and 96% at the experiment 3. By the combination of rule DB and user own rules of scenario-specific rules our method can contribute to detect errors with high ratio.

As for rule description, we will introduce the logical operations, namely AND, OR, NOT among rules and enhance the checking method. Such operations improve the readability of rules and the verifiability of scenarios. If there exist legitimate orderings of events, we can easily and simply specify rules with OR operation.

The quality and the quantity of rules depend strongly on the quality of verification. Although we assume that we

can get rules by a domain analysis, it may be difficult to gather rules for a certain domain. Giannakopoulou et al. propose frameworks for assume-guarantee reasoning [7]. Their method does not require human input in defining assumptions, but generates assumptions automatically. We should consider an automatic generation of rules. The idea of pattern of rules [4] may contribute to generate rules from scenario. In this research, rules are classified into three patterns, namely Rights, Obligations, and Constraints. By analyzing actor, verb, modal auxiliary and conditions in a sentence, categorized rules can be automatically derived from software documents. We have another research plan to generate scenarios with abstract scenario patterns, such as confirm customers, checking out rental items. Since the combinations of these abstract scenario pattern can be regarded as rules, we aim to establish a generating method of rules with pattern based scenario generation.

We can also consider the generic pattern reported, similar to generic patterns and categories of problem domain reported in the CREWS project [10]–[12] and scenario patterns by Ridao et al. [18]. Such patterns will make rules cost-effective and reveal interesting insights about the use of scenarios for requirements specification.

Acknowledgements

The author thanks to Dr. Honghui Zhang at Shimadzu Business Systems Corporation, Japan for his contribution to this research. The author also thanks to Mr. Tatsuya Toyama, Dr. Hiroya Itoga, Mr. Erwin Widodo and other members of Software Engineering Laboratory, Department of Computer Science, Ritsumeikan University, Japan. This research is partly supported by Grant-in-Aid for Scientific Research (C)(2)(17500026), Japan Society for the Promotion of Science.

References

[1] C.B. Achour, "Writing and correcting textual scenarios for system design," Proc. 9th IEEE International Workshop on Database and Expert Systems Applications (DEXA'98), pp.166–179, Vienna, Austria, Aug. 1998.

[2] C.B. Achour, "Guiding scenario authoring," Proc. Eight European-Japanese Conference on Information Modeling and Knowledge Bases, pp.181–200, Finland, 1998.

[3] R. Barish, ACM Conference Committee Job Description, Conference Manual, Section No. 6.1.1, http://www.acm.org/sig_volunteer_info/conference_manual/6-1-1PC.HTM, 1997.

[4] T.D. Breaux and A.I. Antón, "Mining rule semantics to understand legislative compliance," Proc. ACM Workshop on Privacy in the Electronic Society (WPES'05), Alexandria, VA, USA, Nov. 2005.

[5] A. Cockburn, Writing Effective Use Cases, Addison-Wesley, U.S.A., 2001.

[6] C.J. Fillmore, "The case for case," in Universals in Linguistic Theory, ed. E. Bach and R. Harms, pp.1–88, Holt, Rinehart and Winston, New York, 1968.

[7] D. Giannakopoulou, C.S. Pasareanu, and J.M. Cobleigh, "Assume-guarantee verification of source code with design-level assumptions," Proc. 26th IEEE International Conference on Software Engineering (ICSE'04), pp.211–220, Edinburgh, Scotland, May 2004.

[8] M. Jackson, "Problems and requirements," Proc. 2nd IEEE International Symposium on Requirements Engineering (RE'95), pp.2–8, York, England, March 1995.

[9] J.C.S.P. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad, and A. Oloveros, "Enhancing a requirements baseline with scenarios," Proc. 3rd IEEE International Symposium on Requirements Engineering (RE'97), pp.44–53, Annapolis, Maryland, Jan. 1997.

[10] N.A.M. Maiden, "CREWS-SAVRE: Scenarios for acquiring and validating requirements," J. Automated Software Engineering, vol.5, pp.419–446, 1998.

[11] N.A.M. Maiden, M. Cisse, H. Perez, and D. Manuel, "CREWS validation frames: Patterns for validating systems requirements," Proc. Fourth International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'98), Pisa, Italy, June 1998.

[12] N.A.M. Maiden and M. Hare, "Problem domain categories in requirements engineering," International Journal of Human-Computer Studies, vol.49, pp.281–304, 1998.

[13] A. Ohnishi, "Software requirements specification database based on requirements frame model," Proc. IEEE Second International Conference on Requirements Engineering (ICRE'96), pp.221–228, Colorado Springs, Colorado, April 1996.

[14] A. Ohnishi and C. Potts, "Grounding scenarios in frame-based action semantics," Proc. 7th International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'01), pp.177–182, Interlaken, Switzerland, June 2001.

[15] A. Ohnishi, H. Zhang, and H. Fujimoto, "Transformation and integration method of scenarios," Proc. 26th IEEE International Computer Software and Applications Conference (COMPSAC'02), pp.224–229, Oxford, England, Aug. 2002.

[16] Railway Information System Co., Ltd., "JR system," http://www.jrs.co.jp/keiki/en/index_main.html, 2001.

[17] J. Ralyte and C.B. Achour, "Scenario integration into requirements engineering methods," Proc. Workshop on the Many Facets of Process Engineering (MFPE'97), Gammarth, Tunis, Sept. 1997.

[18] M. Ridao, J. Doorn, and J.C.S.P. Leite, "Domain independent regularities in scenarios," Proc. Fifth IEEE International Symposium on Requirements Engineering (RE '01), pp.120–127, Toronto, Canada, Aug. 2001.

[19] A.G. Sutcliffe and M. Ryan, "Experience with SCRAM, a scenario requirements analysis method," Proc. 3rd IEEE International Conference on Requirements Engineering (ICRE'98), pp.164–171, Colorado Springs, Colorado, April 1998.

[20] A.G. Sutcliffe, N.A.M. Maiden, S. Minocha, and D. Manuel, "Supporting scenario-based requirements engineering," IEEE Trans. Softw. Eng., vol.24, no.12, pp.1072–1088, Dec. 1998.

[21] T. Toyama and A. Ohnishi, "Rule-based verification of scenarios with pre-conditions and post-conditions," Proc. 13th IEEE International Conference on Requirements Engineering (RE'05), pp.319–328, Paris, France, Aug.–Sept. 2005.

[22] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenarios in system development: Current practice," IEEE Softw., vol.15, no.2, pp.34–45, March/April 1998.

[23] H. Zhang and A. Ohnishi, "Transformation method of scenarios from different viewpoints," Proc. 11th IEEE Asia-Pacific Software Engineering Conference (APSEC2004), pp.492–501, Busan, Korea, Nov.–Dec. 2004.

Atsushi Ohnishi received B. of Engineering, M. of Engineering, and Dr. of Engineering degrees from Kyoto University in 1979, 1981, and 1988, respectively. He was a Research Associate of Kyoto University from 1983 to 1989 and an Associate Professor of Kyoto University from 1989 to 1994. Since 1994 he has been a Professor at Department of Computer Science, Ritsumeikan University. He was a visiting scientist at UC Santa Barbara from 1990 to 1991 and also a visiting scientist at Georgia Institute of Technology in 2000. His current research interests include requirements engineering, scenario analysis, object oriented analysis, software design techniques, and sequence control programming techniques. Dr. Ohnishi is a member of IEEE Computer Society, ACM, Information Processing Society (IPS) Japan, and Japan Society for Software Science and Technology (JSSST).