1. Introduction

Scenario-based approach is one of the promising techniques not only to elicit requirements from stakeholders, but also to achieve user-participatory development. The techniques to construct scenarios of high quality with less effort are necessary. In addition, since during developing a system its requirements are frequently changed by customers and/or users, the techniques to manage the changes to the scenarios and to handle the change propagations on the scenarios are also crucial ones.

This paper proposes a solution to the above issues based on reusable pattern techniques and case grammar of natural-language processing techniques. We abstract case frames as reusable primitive parts to write sentences in a scenario and patterns to combine the case frames so that we can get consistent scenario episodes.

2. Basic Idea on Scenario Patterns

In our approach, a case frame is an atomic unit to be reused for composing sentences in a scenario. We collect the sentences that frequently appear in a certain problem domain and abstract them to case frames. A scenario developer chooses a suitable set of case frames and evolve it to the episode that he wants. When he evolves it, he adds several sentences to it, and the sentences that can be added are restricted to keep semantic consistency in the episode. Suppose that he constructs a scenario for traveling. Figure 1 illustrates a snapshot on how he constructs it by combining the case frames.

First of all, an actor (traveler in this example) decides where he will travel, and during the travel he stays at the accommodation that he determined in advance. To make this episode, the developer instantiates the two case frames [determine; actor, destination] and [use; actor, object] with words, and line them up in this order. For instance, the word “accommodation” is assigned to the "object" case slot. After instantiating them, the scenario developer gets the following episode:

1) A traveler makes a plan where he will go.
2) He stays at an accommodation.

Consider adding an action of reserving an accommodation in advance and evolving the above episode. The developer instantiates the case frame [reserve; actor, object] and tries to insert it. However this sentence should be put above the action of using the accommodation to make the new episode meaningful because we should reserve the accommodation before we stay at it and the reservation after staying there is meaningless. That is to say, there is a constraint on order of lining the case frames up, and the case frame [reserve; actor, object] should be executed before [use; actor, object]. We should have some constraints on combining case frames and make reusable parts of sets of case frames together with these constraints.

In addition to the order constraint in this example, there is another type of constraints expressing that a certain set of case frames should contain specific case frames. In the example of Figure 1, the simultaneous existence of the frames [determine; actor, destination] and [use; actor, object] in an episode is mandatory. We group the case frames that are mandatory in a meaningful fragment of an episode into an episode framework. Although the case frame [reserve; actor, object] is optional for the traveling episode, the positions where it can be inserted in the episode are restricted with its order con-
constraints. We model case frames, episode frameworks and the constraints on them as reusable parts. We call these parts scenario patterns.

Figure 2 shows the process for constructing a scenario by our reuse approach, and it also includes the representation of the constraints on the case frames. This example is from rental business domain. In the figure, the predicate Req stands for an existence constraint on the frames, while an order constraint is depicted with a directed graph. The predicate Req(O7, O11) specifies that if the case frame O7 is included in the episode, O11 should be included. The case frame O7 stands for the borrower's action of paying a rental fee to the lender, while O11 does for the lender's action of requesting the customer to pay the rental fee.

The developer selects an episode framework consisting of the frames S1 and S2. The frame S2 denotes that the lender really lends the goods that the borrower requests. Therefore the episode framework consisting of S1 and S2 expresses that the customers decides what he wants to borrow and then he get it, and it leads to one of the most primitive episodes. After instantiating it to a primitive episode, the scenario developer begins with the evolution by inserting the case frame O7 to add the action sequence related to the payment of the rental fee. The existence constraint on O7 suggests to him that the frame O11 should be added, and the order constraint forces him to put O7 between S1 and S2 if he adds it. If no payment for the selected rental goods is necessary, he does not need adding O7 and O11. In this sense, these frames are optional to the episode framework. Avoiding the violations of the constraints, he refines and evolves the initial primitive episode to a more complete one by adding the case frames that are specified as the candidates in the reusable database.

3. Experiments and Results

To assess our approach, we picked up a rental business domain as an example and constructed the scenarios of the several applications belonging to the domain. The applications that we used for extracting the scenario patterns were a rental video shop, a library with university laboratory size and a rental costumer, while our subject has constructed the scenarios of the business for a car rental agency by using the extracted reusable patterns. The number of the extracted scenario patterns is 7, and they totally include 42 case frames and 64 constraints. By reusing the extracted patterns, our subject could successfully construct the scenario of the rental car agency business with 78% sentences of them being reused. Scenario patterns in the other domain, automating the extraction of scenario patterns and supporting tool are our future work.

References