Interaction-Based Scenario Integration

Rabeb Mizouni[1], Aziz Salah[2], Rachida Dssouli[1], and Benoît Parreaux[3]

[1] Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3H 1M8, Canada
{mizouni, dssouli}@ece.concordia.ca
[2] Université de Québec A Montréal
Montreal, Quebec, Canada
salah.aziz@uqam.ca
[3] France Telecom R & D, Lannion, France
benoit.parreaux@francetelecom.com

Abstract. *We propose in this paper an interaction-based approach for use cases integration. It consists of composing use cases automatically with respect to interactions specified among them. To this end, we define a state-based pattern for each of these interactions. Then we synthesize a use case interaction graph, which serves the detection of not only unspecified, but also implied use case invocations. Additional constraints are added to the system in order to remove such illicit interactions, called interferences.*

1 Introduction

Due to the rapid growth of distributed systems and their complexity, the production of a system specification has become an arduous task. An incremental elaboration of partial behaviors seems to be more adapted for the requirement elicitation process. Consequently, scenario-based approaches have emerged and rapidly gained on popularity. Rather than specifying the system as a whole, scenario-based approaches consists on describing the behavior as a set of use cases. Each use case depicts a certain functionality of the system and represents its execution traces, defined as scenarios.

Use cases play an important role in the lifecycle process. They can be used in the requirement elicitation phase, as well as in the design, the code generation, and the validation phases. The level of formality in use case notations is strongly dependent on the phase of the lifecycle. During the requirement elicitation phase, use cases are usually described in a friendly notation understandable by the stakeholders. Later on in the lifecycle process, these scenarios are used to synthesize a state-based specification, a more suitable notation for code generation, validation, and verification phase. However, this task is not straightforward. Use cases have to be translated and composed to obtain a formal system specification. In most cases, interactions between use cases do exist. They are either explicitly specified with composition operators, or implicitly detected by a state-based identification. However, in both cases, the resulting state-based model may contain extra unspecified behaviors, called implied scenarios. They are due (in part) to unpredicted use case interactions.

We present in this paper an explicit integration approach for composing uses cases based on their *interactions*. The main objective is the automatic synthesis of an automaton of the system without introducing implied scenarios resulting from the use case merging . An *interaction* represents an invocation of a use case by another. Given a set of use cases, we first translate them into use case automata where edges are labeled with both actions and interactions. Using state-based patterns, the interactions specified within the use cases are translated into a state-based model in order to connect together the respective automata of the interacting use cases. In this context, a pattern defines the state-based representation of a use case interaction. In a second step, we build from the system use cases a *use case interaction graph*. It is used to detect the potential implied use case interactions, called interferences. Additional constraints are added to the system automaton in order to eliminate such interferences.

The paper is structured as follows. In Section 2, we overview the use case model we are using. Section 3 describes the methodology of synthesizing the overall automaton of the system based on the interactions between use cases. Discussions on some related work as well as future work are given in Section 4.

2 Scenario Acquisition : Model Presentation

We have defined in [5] a formal action-based use case representation. Each action expresses the system state evolution. The observation of an action is controlled by a precondition and a postcondition on variables. The approach in [5] allows the synthesis from use cases of an extended automaton which has the behavior specified by a given collection of use cases. When the extended automaton results from only one use case, it is called the use case automaton.

An extended automaton behaves like a classical one, except that firing a transition is enabled by a precondition. Afterward, the system moves to the target location of the transition, and variables values of the automaton are modified according to the assignment of the transition.

In oder to describe the functional requirements of a system, the designer needs to specify a set of use cases. In most cases, interactions between use cases do exist. Offering a notation that gives the possibility to express these interactions promotes not only the reuse, main feature of scenario-based approaches, but also the simplification of the use cases. In fact, when a specific treatment is needed by different use cases, a separate use case can be specified to achieve this task. This use case will be invoked within the other use cases each time the treatment is required, avoiding, in such way, the redundancy in the specification. In this paper, we present three of the most needed interactions between use cases, namely `Include`, `Extend`, and `Interrupt`.

– `Include`(uc_i) specifies a mandatory invocation of a use case uc_i . It models the factorization of a part of the description that is used by many other use cases. If the invocation is done at the end of a use case, `Include` expresses simply the sequential concatenation.

– `Extend(`uc_i`,`$cond_i$`)` specifies an optional invocation of a use case uc_i. Hence, uc_i is only executed when $cond_i$ is verified.
– `Interrupt(`uc_i`, `$cond_i$`)` specifies an interrupting use case uc_i. It is used to state explicitly the actions where an interruption of the current use case can be performed. Unlike the previous relationships, `Interrupt` does not allow the continuation of the interrupted use case after the execution of the interrupting one.

An interaction between use cases is expressed in the calling use case in the form $(a, interaction)$ where $interaction \in \{$ `Include(`uc_i`)`, `Interrupt(`$uc_i, cond_i$`)`, `Extend(`$uc_i, cond_i$`)` $\}$, and a represents the action preceding the interaction. We draw the attention that one may see that `Include(`uc_i`)` could be simply written as an unconditional `Extend` like `Extend(`$uc_i, true$`)`. However, as it will be shown later on, the integration patterns of these two operators in the state-based model are not the same and represent two different semantics. In addition, since scenarios have the objective to facilitate the production of a formal specification from the requirements, it is necessary that the language provides features that are as close as possible to the vision the user has.

3 Integration Approach

We define a two-step automated approach to produce from a set of use cases an automaton modeling the behavior of the overall system. The first step consists of translating the use case interactions into a state-based model. It is achieved by means of state-based pattern for each kind of invocations. Hence, the structure of the obtained automaton reflects the specified interactions. The second step consists of deriving a use case interaction graph, used to detect the induced but undesired interferences between use cases. New control variables are consequently added to the automaton to prevent such behaviors.

3.1 State-Based Interaction Patterns

Interaction patterns serve the automated merge of the use case automata into an overall automaton of the system. They will provide the connection points between the use case automata during their integration. Figure 1 shows the integration pattern of `Include`, `Extend` and `Interrupt` invocations. Transformations are applied to the automaton of the calling use case: transitions are added to refer to the initial location in the called use case. During this process, non-determinism may be introduced because the execution of `Extend` and `Interrupt` calls is conditional. The duplication of the action (for instance action a in Figure 1.(b) and Figure 1.(c)) is inevitable, bringing non-determinism in the automaton.

$start\_uc$ and $return\_uc$ are two new special labels used to connect use cases together. The condition given within an `Extend` or an `interrupt` interactions

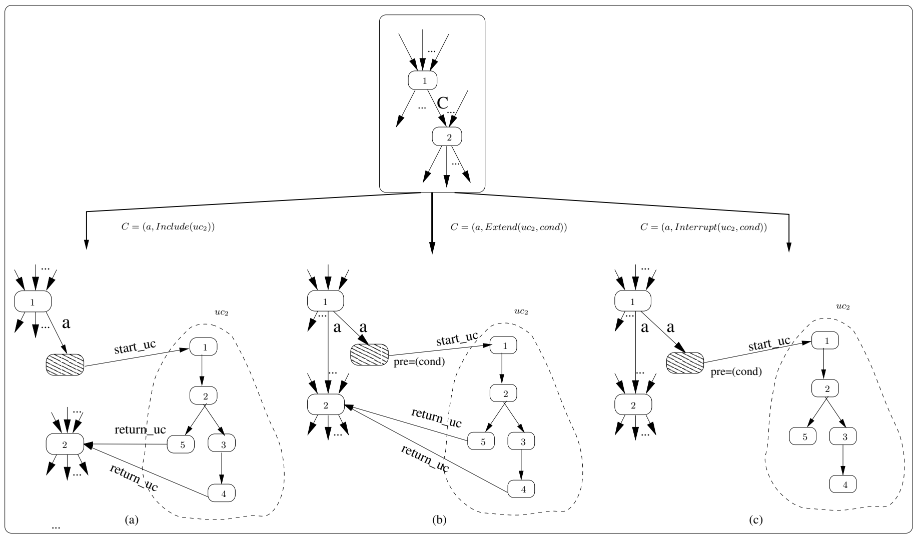

Fig. 1. State-Based Patterns of use case Interactions

is a precondition of the added transition *start_uc*. The actions labeled with *start_uc* and *return_uc* may be decorated afterward with additional pre and post conditions according to the use case interaction graph as we will show later on.

The derived automaton after interpretation of the use case interaction is called *Interaction-Free* use case automaton. Its edges are no more labeled by invocations.

3.2 Use Case Interaction Graph

When specifying the use cases, the designer has defined different interactions. However, since it is difficult that she/he draws the big picture, it could happen that the specified use case interactions generate interferences. The latter may lead to unexpected scenarios in the overall system behavior. Let's consider the case of two use cases making an `Include` invocation to the same use case (c.f. Figure 2 (a) and (b)). After the application of the state-based pattern, the system may eventually run through (a, uc_1, d). The same anomaly appears when a use case makes multiple calls to the same use case (c.f. Figure 2) in different places through the use case. In order to detect such situations, we synthesize a use case interaction graph in which a node is associated to each use case. An edge links together the nodes of two use cases if they interact. The edges are labeled with the type of the interactions as well as the number of their occurrences. The label $(Include, 2)$ in Figure (2 (c)) indicates that the call of the use case uc_1 occurs twice in the description of uc_2.

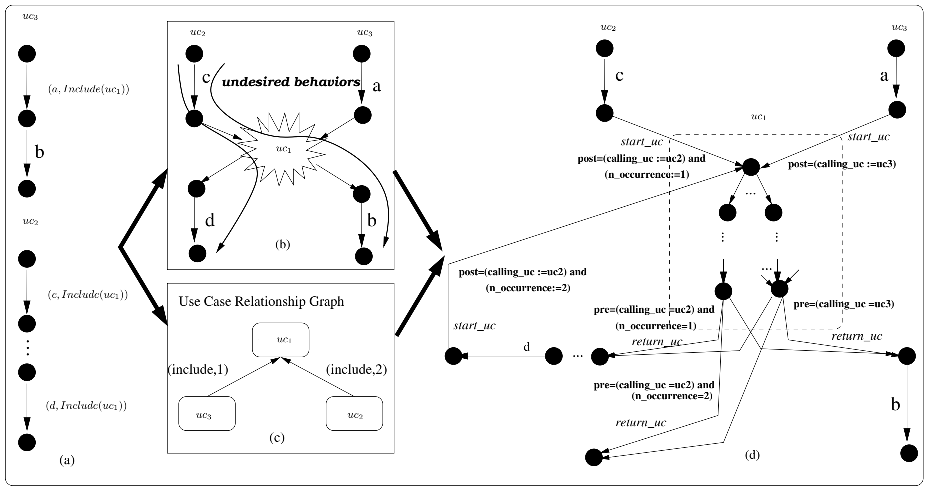

Fig. 2. Inter and Intra-Implied Scenarios Resulting from Use case Merging

The use case interaction graph helps to detect undesired interferences. We distinguish two cases where control variables have to be added to the automaton of the system in order to prevent implied behavior:

– if a use case is referred by many distinct use cases, independently from the type of the interaction, inter-implied scenarios may occur. A new variable `calling_uc` is added to the automaton, and is set by transition *start_uc* to the identifier of the calling use case (c.f. Figure (2 (d)). The *return_uc* transition will be checking the value of the `calling_uc` variable.
– if a use case is referred more than once within the same use case, intra-implied scenarios may occur. A new variable `n_occurrence` is added to the automaton, and is set by the transition, *start_uc*, to the number of the occurrence in the calling use case. In the same way, the *return_uc* transition will be checking the value of the `n_occurrence` variable.

Interaction `Interrupt` makes exception to the previous rules: if a use case is only called through `Interrupt`, no implied behavior is added as there is no return. The number of additional variables added to prevent implied scenarios depend on the existing interactions between the original use cases.

The use case interaction graph serves the generation of the automaton of the system with respect to use case interactions. However, we believe that it could be also used to detect potential problematic interactions. Situations like a use case calling itself, or a mutual calls between two use cases may lead to a disconnected or non-executable parts of the automaton of the system. If the automaton of a use case is disconnected within the automaton of the overall system, it indicates an eventual omission of some use cases or/and interactions, or errors.

At this stage, the merge of the Interaction-Free use case automata represents the automaton of the system. Such automaton may be non-deterministic. The initial locations of the automaton of the system need to be specified by the user.

4 Related Work and Discussion

Many approaches have been developed to synthesize, either automatically or not, state-based models from a set of use cases [1]. State-based models are basically needed to verify and validate the user requirements in order to detect as soon as possible design problems. We brought in this paper the issue of automatic generation of the system automaton based on use case interactions.

Many notations have emerged with different degrees of expressiveness to specify the use cases and their composition. Glinz [4] uses statecharts to model formally scenarios. The integration of scenarios is done in a way to retrieve the relationship between scenarios by keeping their internal structure unchanged, and to detect inconsistencies. The approach proposed carries only the composition of disjoint scenarios with elementary constructors (sequential, alternative, iteration and concurrency constructor). As an extension of this work, Ryser [7] introduces a new kind of charts and notations to model dependencies among scenarios.The advantage of this approach is the fact of defining a notation that captures clearly these inter-scenarios dependencies. Yet, this work is not presenting a methodology rather than a notation that can clarify the dependencies between different scenarios.

In the same range of ideas, UML [6] use case diagrams offer a notation to specify use case relationships. They are similar to the use case interaction graph presented in this paper, but they are more abstract. The labeling of the edge in our case is offering information about the number of occurrence of the interaction through the calling use case, which is a different notion from the multiplicity of a use case relationship in UML2.0. Nevertheless, we can easily extend our model to handle such feature because it already supports control variables on the transition.

Araujo *et al.* [2] focused on representing aspects during use case modeling. They proposed to differentiate between aspectual and non-aspectual scenarios. Similar to our approach, the integration is done on the state machine level. The relationships between use cases are defined in terms of role. In our case, we focus in addition on the automatic generation of the system specification without introducing implied scenarios. To this end, we introduce the notion of use case interaction graph synthesized from the given set of use cases. Bordeleau *et al.* [3] have proposed integration patterns for scenario dependencies. UCMs are used to detect dependencies between scenarios. A state-based specification per use case is then generated for each component and integrated to reflect the scenarios dependencies. The whole process is done manually and relies on the creativity of the designer to connect together the different statecharts in the right way.

Our approach defines a methodology to integrate use cases that consists of (1) developing state-based patterns for the different possible interactions between use cases, and (2) analyzing the relationships between use cases in order to avoid unspecified behaviors. When interferences are detected, they are prevented by adding control variables. The number of added variables is determined according to the use case interaction graph. Hence, the obtained automaton of the system will be as much as possible conform to the original use cases. Finally, our approach promote scalability which is an important feature for industrial applications. The support offered by our method makes the designer do not worry about the overall picture of the system structure when adding new functionalities. As a future work, we aim to develop techniques to minimize the number of these variables. We have presented in this paper three possible interactions, `Include`, `Extend`, and `Interrupt` to illustrate our approach, but our methodology is not restricted to them. Other patterns can be similarly developed.

References

1. D. Amyot and A. Eberlein. An evaluation of scenario notations for telecommunication systems development, 2001.
2. J. Araujo, W. Whittle, and D. Kim. Modeling and composing scenario-based requirements with aspects. In *The 12th IEEE International Requirements Engineering Conference (RE 2004),*, Kyoto, Japan, September 2004.
3. F. Bordeleau and J. P. Corriveau. On the importance of inter-scenario relationships in hierarchical state machine design. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 156–170, London, UK, 2001. Springer-Verlag.
4. M. Glinz. An integrated formal model of scenarios based on statecharts. In *In Schäfer, W. and Botella, P. (eds.) (1995). Software Engineering - ESEC '95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. Berlin, etc.: Springer (Lecture Notes in Computer Science 989).*, pages 254–271, 1995.
5. R. Mizouni, A. Salah, R. Dssouli, and B. Parreaux. Integrating use cases with explcit loops. In *In Proceedings of Nouvelles TEchnnologies de la RÉpartition (NOTERE'04), Saidia, Marocco*, June 2004.
6. J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual.* The (2nd Edition) (Addison-Wesley Object Technology Series).
7. J. Ryser and M. Glinz. Dependency Charts as a Means to Model Inter-Scenario Dependencies. In *In G. Engels, A. Oberweis and A. Zündorf (eds.): Modellierung 2001. GI-Workshop, Bad Lippspringe, Germany. GI-Edition - Lecture Notes in Informatics*, volume P-1, pages 71–80, 2001.