SPECIFICATION SYNTHESIS by MERGING USE CASES*

Aziz Salah
Département d'Informatique
Université du Québec à Montréal
C.P. 8888 succ. centre ville Montréal
Québec Canada H3C 3P8
Email: salah@info.uqam.ca

Rachida Dssouli
Electrical & Computer Engineering
Concordia University
1455 de Maisonneuve Blvd. W. Montreal
Quebec Canada H3G 1M8
Email: dssouli@ece.concordia.ca

Abstract

We propose two methods for automatic merging of use cases to synthesize a specification in the form of a Finite State Machine (FSM) which has the same behavior as these use cases. For each use case the analyst chooses the appropriate method to merge it into the current FSM. The obtained FSM is independent of the order in which use cases are merged.

1 INTRODUCTION

Our objective is the automatic synthesis of a formal specification by merging together a given set of use cases which describes the behavior of the system. Analysts establish use cases and present them to users for validation. Use cases are intelligible for users who are probably not a computer specialist. A use case describes the possible sequences of interactions among the system and its environment in response to a stimulus [1]. It is not a single scenario but regroups a set of potential scenarios. A scenario represents a possible execution of a use case. Usually, a use case focuses on sequences of interactions to accomplish a potential goal. For example, a use case may describe how a phone user may make a phone call. Merging together use cases means the integration of use cases into a single target model which is the specification of the system. In our approach, the output specification is a finite state machine.

In this paper Section 2 describes the acquisition of requirements and their representation. Section 3 presents our methods to merge use cases. Section 4 addresses the related work.

2 ACQUISITION OF THE REQUIREMENTS

Variables of the system are discrete variables which capture the state of the system. When a system interacts with its environment, its state changes. Therefore, the variables of the system are updated to describe its new state. The state of the system depends only on the values of its variables. Thus a state of the system is a configuration in which each variable has a unique value. The value of a variable v in a state s is written $s(v)$. A *variable-constraint* is a boolean combination of predicates on variables. A state of the system satisfies a variable-constraint if the values of the variables in this state satisfy the boolean expression of the variable-constraint. *Variable-assignments* are used to update the state of the system. For example, variable-assignment $Assign = \{v_1 = v_1 + 3, v_2 = 1\}$ means that variable v_1 is incremented by 3, variable v_2 is set to 1 and other variables are unchanged.

Use cases

A use case is described as a tree of possible sequences of *actions*. Figure 1 shows an example of the tree of a use case which specifies the establishment of a phone call. Three scenarios are possible during the execution of the use case shown in figure 1. Let's name these scenarios: sc_1 ="$Dial\_B$, $Ring$, B_Pickup, $Talk$", sc_2 ="$Dial\_B$, $Busy\_Tone\_A$" and sc_3 ="$Dial\_B$, $Ring$, $Busy\_Tone\_A$". sc_1 represents the scenario of the expected actions of the use case and corresponds to establishing a phone call among a user A and a user B. Scenarios sc_2 et sc_3 cover cases of exceptions when a call establishment fails.

We draw the general pattern of the tree of a use case in figure 2 where PN_0, PN_1, .., PN_n are called *primary nodes* and where terms $PAct_i$ and $EAct_{ij}$ are *actions*. PN_0 is the root of the tree and represents the starting point of the execution of the use case. All executions of a use case start from the root and end by a leaf of the tree. Actions $PAct_i$ are called primary actions of the use case because the scenario "$PAct_1$, $PAct_2$, .., $PAct_n$" represents the expected sequence of actions of the use case. Actions $EAct_{ij}$ are exception and represent either a timer expiration, an

*Partially supported by France Telecom

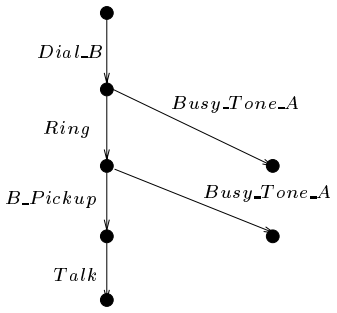

Figure 1: Example of a use case modeling the establishment of a phone call

interruption or any problem preventing the use case from ending the execution of the expected sequence of actions. Actually, scenarios in the form "$PAct_1$, $PAct_2$, .., $PAct_i$, $EAct_{ij}$" represent the cases where the execution of the use case fails. The tree of a use case shows the causality relation among its actions. In the sequence of a scenario, the execution of an action creates a *context* which enables the execution of the next action. Before characterizing such *context*, let's focus on the description of an action.

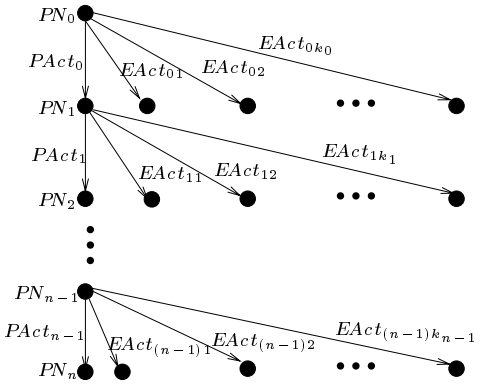

Figure 2: Representation of the tree of a use case

In a use case, the description of an action is a structure which includes three components: an event, a pre-condition and a post-condition. The event is a label belonging to Lab the set of possible events of the system and models an observable signal. The pre-condition is a variable-constraint that the state of the system must satisfy before the execution of the action. The post-condition of an action is a variable-assignment which describes the modification of the state of the system after the execution of the action. Moreover, we write $Act.Label$ the event of action Act, $Act.VarConst$ denotes its variable-constraint and $Act.Assign$ represents its variable-assignment. In the next section, we define a formal semantics of the behavior a use case describes.

The merging of a use case into the specification is accomplished by the means of an integration method. We propose two semanticly different integration methods; namely $FreeMethod$ and $BlockMethod$. $FreeMethod$ enables scenarios interleaving in the specification contrary to $BlockMethod$ which considers a use case as a building block and protects it against interleaving with other use cases. The analyst chooses the appropriate method to use for each use case and represents this information within a mapping $IntegMethod$. Finally, We assume that the outputs of the requirements acquisition phase are:

- a set of discrete variables $V = \{v_1, .., v_p\}$
- a set of use cases $UC = \{uc_1, uc_2, .., uc_q\}$
- and the mapping $IntegMethod$ which indicates for each use case in UC an integration method.

3 SYNTHESIS OF THE SPECIFICATION

Synthesizing a specification means merging all the given use cases into a finite state machine which preserves the behavior of use cases. We start by defining a formal semantics for use cases then we transform use cases into a flat form which is appropriate to use for synthesizing a finite state machine.

3.1 Formalization of the semantics of a $FreeMethod$ use case

A $FreeMethod$ use case is a use case which is merged into the specification by using $FreeMethod$. In a scenario, the execution of action $PAct_i$ (figure 2) creates a *context* that enables the execution of the next actions. All actions of the primary node PN_{i+1} are candidate to be next action after $PAct_i$. Consequently, we identify this *context* by a variable-constraint written "$PN_{i+1}.VarConst$" and associated with PN_{i+1}. The state of the system should satisfy the context variable-constraint $PN_i.VarConst$ as well as $Act.VarConst$ in order to enable the execution an action Act of a primary node PN_i. We notice that the context $PN_0.VarConst$ is propagated through the execution of action $PAct_0$ to constitute the context $PN_1.VarConst$. In the same way, the context $PN_1.VarConst$ is propagated through the execution of action $PAct_2$ to constitute the context $PN_2.VarConst$ and so on. Consequently, the variable-constraint of the $context$ of a primary node is deduced from a recurrent sequence:

$$PN_0.VarConst \stackrel{def}{=} PAct_0.VarConst \wedge (tag == NoUC) \quad (1)$$

For $1 \leq i \leq n-1$,

$$PN_i.VarConst \stackrel{def}{=} PropagateConst(PN_{i-1}, PAct_{i-1}) \wedge PAct_i.VarConst \quad (2)$$

$$PN_n.VarConst \stackrel{def}{=} PropagateConst(PN_{n-1}, PAct_{n-1}) \quad (3)$$

In equation (1) we have introduced a new variable tag which is used during the merge of use cases. It is considered as a variable of the system and added to the set V. The variable tag has no effect in the case of $FreeMethod$ use case but it's needed for $BlocMethod$ use cases. When the variable tag is set to the value $NoUC$, it means that no $BlocMethod$ use case is running.

Equation (1) states that the context $PN_0.VarConst$ is exactly the pre-condition of primary action $PAct_0$. When the state of the system satisfies $PN_0.VarConst$, the execution of $PAct_0$ is expected but the environment of the system may not allow this execution then an exception action of PN_0 may happen provided that the state of the system satisfies also the pre-condition of this exception action.

Equation (2) gives the context expression for all primary nodes except the root and the leaf ones. The term $PropagateConst(PN_{i-1}, PAct_{i-1})$ denotes the variable-constraint that all the states of the system satisfy after the execution of action $PAct_{i-1}$. It represents the variable-constraint which results from the propagation of $PN_{i-1}.VarConst$ by applying $PAct_{i-1}.Assign$. Moreover, we must include $Act_i.VarConst$ as a part of the expression of $PN_i.VarConst$ because an exception action of PN_i may be executed in states where the execution of $PAct_i$ is expected. Consequently, the context $PN_i.VarConst$ is the conjunction among $PropagateConst(PN_{i-1}, PAct_{i-1})$ and $PAct_i.VarConst$.

Equation (3) shows the context expression of the last primary node of a use case. Since this primary node doesn't contain any action, its context is exactly the resulting variable-constraint from the execution of $PAct_{n-1}$.

Let's now define the formal semantics of a $FreeMethod$ use case. Given an action Act of a primary node PN of a such use case, the system enables the execution of Act in a state s which satisfies the variable-constraint $Act.VarConst \wedge PN.VarConst$. During the execution of Act in s, the label $Act.Label$ is observed then the system moves to the new state which result from the application of the variable-assignment $Act.Assign$ in state s. We will define later the formal semantics of a $BlockMethod$ use case.

3.2 Behavior Rules of a $FreeMethod$ use case

On the one hand, the representation of a use case in the form of a tree (figure 2) is clear and intelligible for users because they can easily extract from given use cases all the possible scenarios of the system behavior. On the other the analyst needs more convenient representation for use cases which is suitable to an automatic synthesis of a specification from given use cases. For this reason we transform a use case into a flat format composed of a set of independent *behavior rules*. A *behavior rule* is obtained by extending an action of a use case with the context of the primary node of this action. A behavior rule has the same description elements like an action but has a different semantics. A behavior rule r has a variable-constraint $r.VarConst$, a label $r.Label$ and a variable-assignment $r.Assign$. Behavior rules are context free which means: if the state of the system satisfies $r.VarConst$ then the system may execute behavior rule r and moves to a new state by applying $r.Assign$ to the current state.

In order to preserve the semantics of a $FreeMethod$ use case (section 3.1), we define the behavior rule r of an action Act of a primary node PN as follows:

- $r.VarConst \stackrel{def}{=} Act.VarConst \wedge PN.VarConst$,
- $r.Label \stackrel{def}{=} Act.Label$ and
- $r.Assign \stackrel{def}{=} Act.Assign$.

3.3 The case of $BlockMethod$ use cases

The normalization procedure (Fig. 3) consists of modifying a $BlockMethod$ use case to obtain an equivalent $FreeMethod$ use case. We bring thus the definition of the formal semantics of a $BlockMethod$ use case to the case of $FreeMethod$ use case. The normalization procedure asserts that the scenarios of a $BlockMethod$ use case are not interruptible by any other use case scenarios. It consists of setting the variable tag to $uc.id$ by modifying the variable-assignment of the the first primary action of uc (Fig.3 line 2). $uc.id$ denotes the unique id of the use case uc. After the execution of the the first primary action of a normalized $BlockMethod$ use case, the variable tag remains set to $uc.id$ until the end of the use case where the value of tag is restored to $NoUc$ by the execution of one of the last actions (Fig.3 lines 3-5). The variable tag is used as a token to disable other use case when a $BlockMethod$ inserted use case is running.

3.4 Finite state machine of behavior rules

Finite state machine (FSM) is a suite model widely used by the computer science community. FSMs are used for the

Procedure $normalization(uc : usecase)$
(1) **If** $IntegMethod(uc) = BlockMethod$ **Then**
(2) $PAct_0.Assign := PAct_0.Assign \cup \{tag = uc.id\}$
(3) $PAct_n.Assign := PAct_n.Assign \cup \{tag = NoUC\}$
(4) **For all** $EAct_{ij}$ **of** uc **Do**
(5) $EAct_{ij}.Assign := EAct_{ij}.Assign \cup \{tag = NoUC\}$

Figure 3: $Normalization$ procedure

specification of the behavior of a system, for the recognition of language patterns etc. FSM are precise and easy to read and to convert into a design and into code [2]. FSM is a directed labeled graph where nodes are states which are connected by labeled edges called transitions. Formally, we write $M = (S, S_o, L, T)$ is an FSM where, S is the set of states, $S_o \subset S$ is the set of initial states, L a set of labels for transitions and $T \subset S \times L \times S$ is the set of transitions. FSM is an abstraction of the behavior of a system which allows its simulation. Firing a transition of an FSM represents the execution of an action.

We aim at constructing an FSM which has the same behavior as a set of behavior rules B. This task is straightforward and consists of constructing the components (S, S_o, L, T) of this FSM. S is the set of states of the system the behavior is specified by B. The states of the system were defined in section 2. T the set of transitions is composed of all the transitions in the form $(s, r.Label, s')$ where r is a behavior rule belonging to B, s is a state of the system which satisfies $r.ConstVar$ and s' is the state which results for the application of $r.Assign$ in s. L is the set of all the labels of the behavior rules in B. However, we will discuss further the definition of S_o the set of initial states because this information is not available in a set of behavior rules and should be extracted from use cases.

We show at Figure 4 a $FreeMethod$ use case and the obtained FSM from its behavior rules. The state $(NoUC, 0, 0)$ is the initial state of this FSM. By generalizing this reasoning, the initial states of the FSM of a set of use cases are the initial states of use cases.

3.5 Merging use cases into FSM

It consists of building an FSM from the behavior rules which are extracted from use cases of UC according to their decided integration methods given by the mapping $IntegMethod$. This FSM is the specification of the system. Other use cases may be added to the specification to complete any lack. The FSM of a system is thus incremently build. Any intermediate FSM can be validated and tested. If any error or bad interaction among use cases is

$Action$	$ConstVar$	$Label$	$Assign$
$PAct_0$	$v_1 == 0 \wedge v_2 == 0$	Out	$v_1 = 1$
$PAct_1$	$True$	In	$v_1 = 0, v_2 = 1$
$EAct_{11}$	$True$	$TimeOut$	$v_1 = 1$
$PAct_2$	$True$	$Reset$	$v_1 = 0, v_2 = 0$

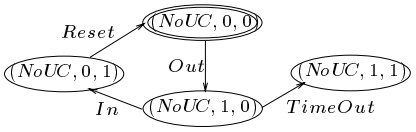

Figure 4: A use case and its FSM where a state is a tuple of variables tag, v_1 and v_2 values

detected, the responsible use cases are traced. After modifying those use cases or choosing for them an another integration method, a new FSM is then rebuild. Since FSM of the system is obtained from a set of behavior rules, we will obtain the same FSM no matter the order in which use case are merged.

We present at Fig. 5 an illustration of merging use case using $FreeMethod$ and $BlockMethod$ integration methods. FSM of figure 5(c) results from the integration of use cases uc_1 (figure 5(a)) and uc_2 (figure 5(b)) using for both of them $FreeMethod$. Because of the overlapping of use cases uc_1 and uc_2, there is in FSM (c) a new scenario $''a, b, g''$ which is not an execution of uc_1 nor uc_2 but results from their interleaving. The scenario "a, b, g" do not occur in FSM figure 5(d) which results from the merging of uc_1 by $BlockMethod$ and uc_2 by $FreeMethod$. The scenario "a, b, c, d" is thus protected from interleaving in FSM (d).

4 RELATED WORK AND DISCUSSION

In the related work, some authors do not differentiate use cases and scenarios. A scenario should be an execution of a use case. A use case with only one possible execution may be considered as a scenario.

Glinz [3] describes his use cases using Harel's state charts formalism [4] for which use cases integration templates were defined. Overlapping use cases must be decomposed into disjoint ones before their integration. Amyot and al. [5] have chosen a UCM *Use Cases Maps* [6] representation for their scenarios. UCM is graphic notation which is not formal but it is suitable for requirements elicitation. Koskimies and al.[7] use scenarios to describe partial behaviors of object classes in the OMT method. Scenarios are formalized as trace diagrams and integrated into a finite state machine that contains exactly the scenarios.

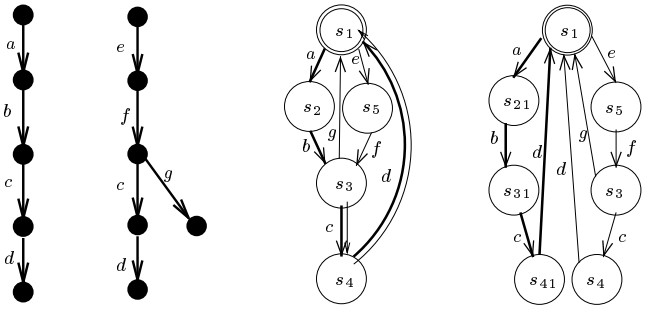

Figure 5: (a) and (b) are the trees of use cases uc_1 and uc_2 respectively. (c) is FSM of the integration of uc_1 and uc_2 using for both of them $FreeMethod$. FSM (d) is obtained when $BlockMethod$ is used for uc_1 and $FreeMethod$ for uc_2

Their integration algorithm is restricted to deterministic and "completely specified" systems. Hsia and al. [8], construct a tree of all possible behaviors by considering at each node all the possible events according to a user view. The scenarios of a user view are paths this tree and are translated into a grammar that corresponds to a finite state machine. The generated specification is complete according to the set of defined events. Somé and al. [9], describes scenarios using restricted natural language. Their scenarios support timed constrained behavior and are integrated into a timed automaton[10].

Merging use case in our approach is completely automatic. We have also proposed two semanticly different integration methods. Choosing one of the integration methods for merging a use case is a part of the specification. Consequently, a specification error may be fixed just by changing the integration methods of use cases. Moreover a similar approach may be applied for real-time systems [11].

5 CONCLUSION

In this paper, we have proposed a suite formal model of use case. An incremental construction of a specification results from merging use cases by using two possible integration methods. Choosing an integration method for a use case is based on whether the use case interleaving it is enabled or not. The output of the integration of use case is an FSM which is insensitive to the order in which use cases are inserted.

Our approach was implemented as a formal specification support tool in which several systems were tested such as a telephone switch, an automatic teller machine and a mouse click recognition system. The number of states in the synthesized FSM grows rapidly, we wish to further study this problem. Our approach may also be applied into Object oriented analysis, in this case each object may represent a system. This is an interesting path we will explore.

REFERENCES

[1] J. Rumbaugh, Getting started, Using use cases to capture requirements, Journal of Object Oriented Programming (1994) 8–??

[2] S. R. Schach, Object-Oriented and Classical Software Engineering, 5th Edition, McGraw-Hill, 2001.

[3] M. Glinz, An integrated formal model of scenarios based on statecharts, in: W. Schäfer, P. Botella (Eds.), Proceedings of the Fifth European Software Engineering Conference, no. 989 in Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 254–271.

[4] D. Harel, STATECHARTS: A Visual Formalism for Complex Systems, Science of Computer Programming 8 (1987) 231–274.

[5] D. Amyot, L. Logrippo, R. Buhr, Spécification et conception de systèmes communicants: une approche rigoureuse basée sur des scénarios d'usage, in: CFIP97, Liège, Belgique, 1997, pp. 159–174.

[6] R. Buhr, R. Casselman, Use Case Maps for Object-Oriented System, Prentice Hall, USA, 1995.

[7] K. Koskimies, E. Mäkinen, Automatic Synthesis of State Machines from Trace Diagrams, Software-Practice and Experience 24 (7) (1994) 643–658.

[8] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen, Formal approach to scenario analysis, IEEE Software 11 (1994) 33–41.

[9] R. Dssouli, S. Some, J. Vaucher, A. Salah, Service creation environment based on scenarios, Information and Software Technology 41 (11-12) (1999) 697–713.

[10] R. Alur, D. Dill, A Theory of Timed Automata, Theoretical Computer Science 126 (1994) 183–235.

[11] A. Salah, Génération automatique d'une spécification formelle à partir de scénarios temps-réels, Phd thesis, Université de Montréal (May 2002).