Composition of Use Cases using Synchronization and Model Checking

R. Mizouni[1], A. Salah[2], S. Kolahi[3], R. Dssouli[1]

[1] Electrical and Computer Engineering Department, Concordia University
{mizouni,dssouli}@encs.concordia.ca
[2] Computer Science Department, UQAM University
aziz.salah@uqam.ca
[3] Computer Science Department, Concordia University
s_kolahi@cs.concordia.ca

Abstract Capturing the behavior of a system by use cases have been intensively investigated in the last decade. The challenge is to find both the adequate model that fits the needs of the analyst and a formal composition mechanism which helps the generation of the expected behavior. In this paper, we propose a formal approach for specifying and composing use cases based on assignments. Those assignments are used to express new use cases. An assignment provides the join points and the composition operators that will be taken into account during the composition. These join points are, in fact, determined through a model checking step. They represent states where a property defined by the analyst holds. In order to evaluate these assignments, we define a composition mechanism based on the well known concept of synchronized product.

Keywords: Use cases, model checking, composition operators, synchronized product

1 Introduction

Capturing the system behaviors within use cases has gained a lot of interest during the last decade. Use cases represent a partial behavior of the system, which helps the requirement elicitation process. However, composing use cases in order to generate the system specification is a challenging task. Its complexity lies within the formality of the model representing use cases, the detection of states on which the composition is performed, and the level of automation of the composition.

Defining interactions among use cases is another challenge for the analyst which may be specified explicitly using composition operators, namely sequential concatenation, iteration, alternative and etc. After the composition according to specified operator semantics, the obtained behavior may not meet the analyst's intended point of view because of possible unexpected interactions. Retrieving unexpected interactions is a hard task which makes the incremental construction of the specification a helpful means for getting the right system behavior.

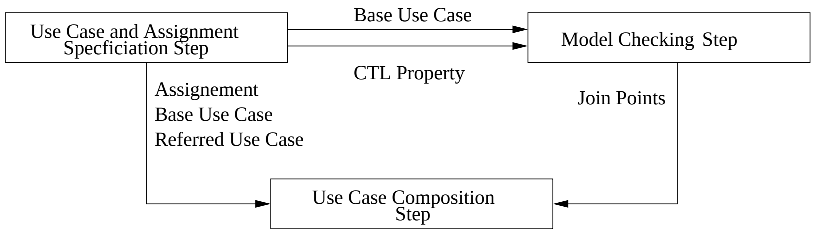


Fig. 1 : Approach Overview

Furthermore, in order to explicitly specify the interactions, the analyst has to choose the states where to compose use cases, called *join points*. This choice again is a hard task that requires deep understanding of the characteristics of each state within the use case. The usage of temporal property for determining these composition join points helps the process of generating the system specification, especially when the size of use cases increases.

This paper addresses a formal, automated and incremental approach for use case composition using *assignments*. The approach consists of three steps, as shown in Fig Fig. 1 : use cases and assignments specification step, a model checking step, and a composition step. First, the analyst provides a set of use cases and a set of assignments. For each assignment, a new use case that represents the evaluation of this assignment is generated. Each assignment uses two use cases: the base use case and the referred use case. The base use case is the one where the new behavior will be added while the referred use case represents the additional behavior to be weaved within the base use case. Moreover, the assignment includes a composition operator and a CTL [1] property which is used to identify the join points. The states of the base use case where the property holds are determined by a model checker, and then, selected as joint points. The composition will be performed on these states respecting the semantics of the composition operator of the assignment. These semantics are achieved by means of the composition based on the synchronization product of two use cases on common labels, as we will show later. The use case that results from the composition represents the evaluation of the assignment.

The paper is structured as follows. In Section 2, we give an overview of the notation we are using in the paper, and in Section 3, we present the definition of assignments. In Section 4, we describe our approach for composing use cases and synthesizing the system automaton. By an example of an invoicing system, Section 5 shows the applicability of our approach to distributed use cases. Discussion of related works is given in Section 6. Finally, we draw our conclusions and discussions on future works in Section 7.

2 Preliminaries

A use case is used to describe a functional behavior of the system regarding a certain concern. The behavior represented as a use case is composed of sequences of actions.

A finite state automaton model is used to express the behavior of a use case because of its expressiveness power and its formality level. A finite state automaton (FSA) is defined as a 5-tuple *(S, s^0, S^f, L, E)*, where S is the set of states, $s^0 \in S$ is the initial state, $S_f \subseteq S$ is the set of final states, L is the set of labels, and $E \subseteq S \times L \times S$ is the set of transitions. For a transition $(s,l,s') \in E$, we write $s \xrightarrow{l} s' \in E$. A clone of a use case is an automaton generated from the use case, with having the same structure and same set of behaviors, but different edge labeling. Next, we present the formal definition of a clone of a use case FSA.

Definition 1 (Clone of a use case)

A clone of use case FSA $A=(S, s^0, S^f, L, E)$ respecting a renaming function *Rename*: $L \rightarrow L'$ is a use case FSA $A'=(S, s^0, S^f, L', E')$ such that:

$$\forall e = (s_1, l, s_2) \in E, \exists e' \in E' \textit{ such that } e' = (s_1, Rename(l), s_2)$$

The clone of a use case is obtained by renaming its different labels using a renaming function.

We will use synchronization for composing FSAs. We present our definition of synchronized product which is based on synchronization at common labels.

Definition 2 (Synchronized product on common labels)

Let $A_i = (S_i, s^0_i, S^f_i, L_i, E_i)$ *for n FSAs*. We define the synchronized product of A_i $i=1..n$ in their common labels as the connected component containing the state $(s^0_1, \ldots, s^0_n)$ of the FSA (S, s^0, S^f, L, E) where $S \subseteq S_1 \times \ldots \times S_n$, $s^0 = (s^0_1, \ldots, s^0_n)$, $S^f \subseteq (S^f_1 \times \ldots \times S_n) \cup (S_1 \times S^f_2 \times \ldots \times S_n) \cup \ldots \cup (S_1 \times S_2 \times \ldots \times S^f_n)$, $L \subseteq (L_1 \cup L_2 \cup \ldots \cup L_n)$, and E is the set of transition defined by the inference rules :

$$\frac{(s_i \xrightarrow{l} s_i' \in E_i), (l \notin L_j, 1 \leq j \neq i \leq n)}{(s_1, s_2, .., s_i, .., s_n) \xrightarrow{l} (s_1, s_2, .., s_i', .., s_n) \in E} \qquad (1)$$

$$\frac{((s_i \xrightarrow{l} s_i' \in E_k), k \in J) \textit{ where } J = \{ j / l \in (\bigcap_{1 \leq j \neq i \leq n} L_j)\}}{((s_1, ..., s_n) \xrightarrow{l} (s_1'', ..., s_n'') \in E), (s_i'' = s_i' \textit{ if } (i \in J)), (s_i'' = s_i \textit{ if } (i \notin J))} \qquad (2)$$

Rule (1) states that when a label belongs to a unique FSA, then only this FSA fires the transition. Rule (2) shows that when a label belong to more than one FSA, then all these FSAs synchronize in order to fire the transition at the same moment.

After specifying the use cases to compose, the analyst has to describe properties. We use the *Computation Tree Logic* (CTL) formalism for its expressiveness to describe both safety and liveness properties of the system in the states. Given a CTL formula φ and a state s, $s \models \varphi$ whenever φ is true in s.

Definition 3 (Join Point Set)

Let $A = (S, s^0, S^f, L, E)$ a use case FSA. Join point Set J of a CTL formula φ in A is a set of states S such that $J=\{s \in S / s \models \varphi \}$.

This set defines the states where the composition will be performed.

3 Assignment Specification

3.1 Use Case Composition Operators

The analyst can specify different operators to model interactions between use cases. The `Include` composition operator specifies that the base use case has to include the behavior of the referred use case during the execution flow in the join point. After the execution of the referred use case, the base use case would resume from the join point. The `Extend_with` composition operator specifies that the behavior of the base use case *may* include the behavior of the referred use case. Again After the execution of the referred use case, the base use case would be resumed from the join point. Finally, the `Interrupt_with` composition operator specifies that the flow of execution of the base use case may be interrupted by the referred use case. In this case, unlike the previous operators, base use case would not be resumed after the execution of the referred one. We are presenting our approach in the case of these three which are the most known operators. However, our approach is not limited to them and the same process can be applied in order to consider other operators such as sequential concatenation.

3.2 Assignment Description

Assignments are used to specify the composition information between two use cases. These assignments are equations used to create new use case FSAs from the existing ones with respect to the semantics of the composition operators. They follow the syntax:

```
Z: = Composition_Operator (X, Y) Where φ
```

`Where Z` represents the FSA that will be generated from the evaluation of the assignment, `X` is the base use case and `Y` is the referred one. `Composition_Operator` represents one of the three specified composition operators, `Include`, `Extend_with`, and `Interrupt_with`. Finally, `Where φ` defines the set of join points where the composition will be performed. As said previously, it is defined by the set of states where the property φ holds.

It is important to note that the composition is performed on states rather than transitions. Contrarily to a transition based composition, a state based composition results in all edges related to that state being affected by the assignment. Furthermore, unlike other approaches such as aspect-oriented approaches, there is no need for the qualifiers *Before* and *After* defined with the join point where the composition is done. In our case, the two expressions “*Before s*” and “*After s*” lead to bisimilar FSAs .

4 Use Case Composition Approach

4.1 Join point Generation

After the definition of the assignment by the analyst, the property as well as the base use case is sent for model checking. As stated before, this property is used to find the set of states on which the composition should be performed. Since model checkers return only true or false with a counterexample, for each state of the base use case starting from the initial one, we run the model checker as if it was the initial state of the base use case. If it returns true then the property holds in that state, if it returns a counter example, then the property does not hold in that state and is not a member of our joint point set. The resulting set would act as the place where the composition should be done.

As a result of the model checking step, the join point set could be empty or not. In case of empty set, the base use case will never verify such property and no new use case can be generated from the evaluation of the assignment. Therefore a revision of either the property or the use cases is needed. On the other hand, when the resulting join point set contains more than one state, the composition of the two use cases should be done in all these states. For that purpose, two approaches can be considered. The first one is to do the composition in an incremental manner. This means that we compose first the two use cases in one state. Then, the resulting use case from the first iteration is used for composition in another state and so on until all the join points are considered. This approach brings the problem of state traceability since the resulting states from the first iteration are no more the states present in the base use case and hence they can not be traced. Moreover the convergence of the approach has to be proved since the synchronized product may duplicate states in the resulting use case. The second solution consists of generating FSAs that takes into account the semantics of the assignment on the different states where the property holds and then applying synchronization on all of them in order to derive the new use case. We present this solution in the next section.

4.2 Composition approach

After retaining the join point set, base and referred use cases have to be composed. From behavioral point of view, the traces of the referred use cases are inserted within the trace of the base use case in all the states of the join point set with respect to the semantics of the operator. In order to achieve this composition, we propose to synthesize a set of FSAs from the use case FSA, which we call *builders*. Each builder reflects the semantics of the composition operator in a join point. Builders would synchronize in order to generate the intended new use case. They are generated automatically from use cases with respect to specific synthesis rules as we will show next. Fig. 2 shows the composition approach. After determining the set of join points, a set of referred use case clones has to be generated by labeling renaming. Next, builders are generated and then composed, resulting in a synchronized product from which we extract an intermediate use case FSA. Finally, we generate the new use case by recovering the original labeling of the referred use case. This new use case is added to the originally specified set of use cases and may be used for describing new assignments. In the next section, we present the formal details of each of these steps.

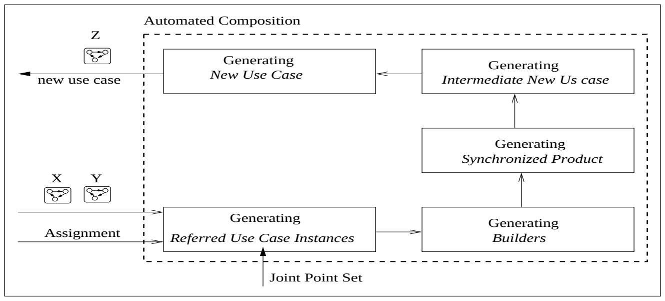

Automated Composition
Z
new use case
Generating New Use Case
Generating Intermediate New Us case
Generating Synchronized Product
X
Y
Assignment
Generating Referred Use Case Instances
Generating Builders
Joint Point Set

Fig. 2: Composition Approach

4.2.1 Clone Synthesis

As mentioned in Definition 1, clones of a use case are generated using a renaming function for relabeling the alphabet of the original use case. In fact, for each join point $s \in J$, a clone of the referred use case has to be generated. This is for two reasons: (1) to differentiate it during the synchronization and hence avoid deadlock caused by common labels (2) to synchronize with the base use case builder generated for composition in state s.

In order to automate the synthesis of the clone FSA, we define a renaming function that modifies the labeling of the FSA. It uses the joint point state where the clone will be considered for composition. Let $A_1 = (S_1, s^0_1, S^f_1, L_1, E_1)$ and $A_2 = (S_2, s^0_2, S^f_2, L_2, E_2)$ two use case FSAs such that A_1 is the base use case and A_2 is the referred one. Let φ be the property specified in the assignment and J the set of join points retained from the model checking phase. The renaming function for state $s \in J$ is:

$$f_s : L_2 \cup \{begin, end\} \rightarrow L_2^s \cup \{begin_s, end_s\} such\ that :$$
$$\forall l \in L_2,\ f_s(l) = l_s$$
$$f_s(begin) = begin_s$$
$$f_s(end) = end_s$$

The labels *begin* and *end* are put during the generation of builders. They are used for synchronization in order to indicate where the referred use case has to be inserted in the base use case. The generated clone of FSA A_2 with the renaming function f_s is the FSA $A^s_{2_{clone}} = (S_2, s^0_2, S^f_2, L^s_2, E^s_2)$.

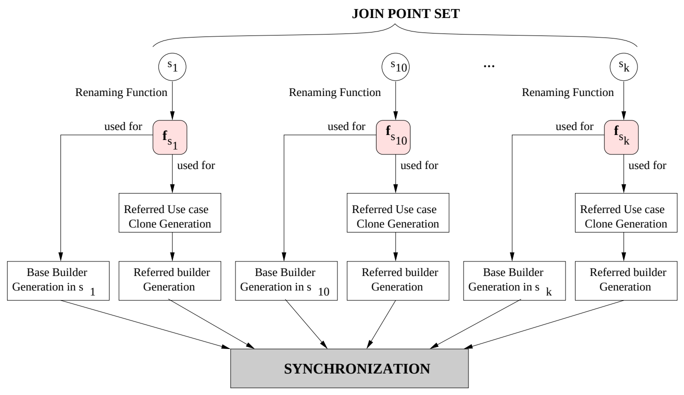

JOIN POINT SET
s1
s10
...
sk
Renaming Function
Renaming Function
Renaming Function
used for
fs1
used for
fs10
used for
fsk
used for
used for
used for
Referred Use case Clone Generation
Referred Use case Clone Generation
Referred Use case Clone Generation
Base Builder Generation in s 1
Referred builder Generation
Base Builder Generation in s 10
Referred builder Generation
Base Builder Generation in s k
Referred builder Generation
SYNCHRONIZATION

Fig. 3: Synthesis of base and referred builders

4.2.2 Base Use Case builders Synthesis

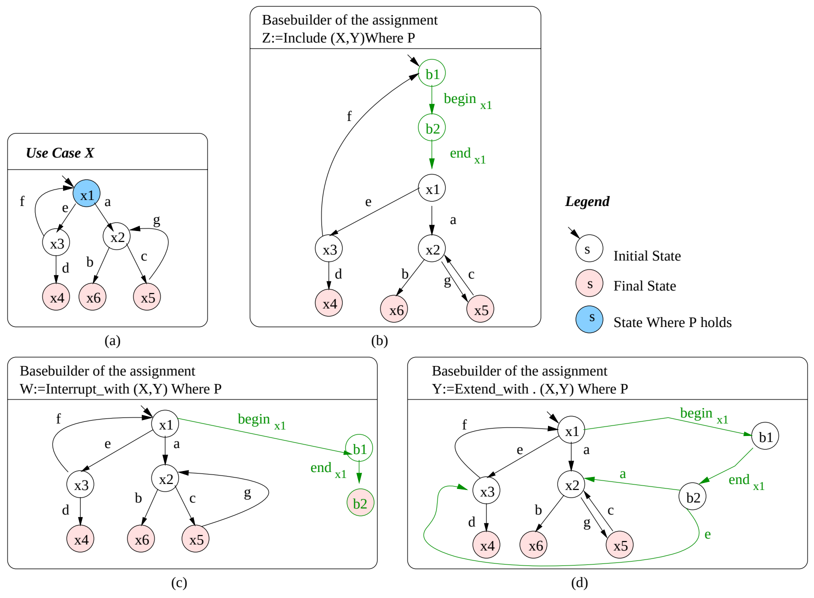
When there is more than one state in the join point set, we end up with a set of base builders, each of them constructed in order to show the insertion of a corresponding referred builder in the join point state, as illustrated in Fig. 3. It is the renaming function f_s which is building this link. In fact, for each state in the set of join points, a clone of the referred use case is created using f_s and a base builder is synthesized to show the insertion of referred use case in the state *s*.

For each s $\in J$, we construct a base builder from the use case A_1 with respect to the renaming function f_s. The synthesized base builder is an FSA $A_1^s = (Q, q^0, Q^f, L \cup \{f_s(begin), f_s(end)\}, T)$ that reflects the semantics of the composition operator as well as the join point *s*. The labels of the base use case are not renamed in the base builder, only two labels f_s *(begin)* and f_s*(end)*) are added which serve as the common label indicating the start and the end of the insertion of the referred use case within the base one. The two builders will synchronize on these labels. We present the set of synthesis rules of the FSA A_1^s in Table 1 for each of the composition operators we defined. These rules are defined for a unique join point *s*.

Let's consider the case of an `Extend_with` composition in the state *s*. The synthesis of the base builder follows the rules (7-11). Rule (7) defines the set of the states of the builder FSA while Rule (8) defines the set of its final states. Rule (9) shows that the labeling of all the transitions that are not outgoing from *s* are labeled with the same label *a*. Rule (10) demonstrates that from the state *s* new added transitions labeled with f_s*(begin)* and f_s*(end)* synchronize with the builder of referred use case clone. Finally, Rule (12) shows that all the outgoing transitions of *s* are duplicated in order to handle resuming of the base use case after the insertion of the referred use case clone. Fig. 4 (d) gives an example of such a base builder.

Table 1: Synthesis Rules of Base builders

Include (X,Y) Where φ (φ holds in the state s)	
$\dfrac{S}{\mathrm{Q} = \mathrm{S} \cup \{\mathrm{q}, \mathrm{q}'\}}$	(3)
$\dfrac{S^{f}}{\mathrm{Q}^{\mathrm{f}} = \mathrm{S}^{\mathrm{f}}}$	(4)
$\dfrac{(x \xrightarrow{a} x' \in E), ((x' \neq s))}{x \xrightarrow{a} x' \in T}$	(5)
$\dfrac{(x \xrightarrow{a} x' \in E), (x' = s)}{(x \xrightarrow{a} q \in T) \in T, (q \xrightarrow{f_s(begin)} q') \in T, (q' \xrightarrow{f_s(end)} x_1) \in T}$	(6)
Extend_with(X,Y) Where φ (φ holds in the state s)	
$\dfrac{S}{\mathrm{Q} = \mathrm{S} \cup \{\mathrm{q}, \mathrm{q}'\}}$	(7)
$\dfrac{S^{f}}{\mathrm{Q}^{\mathrm{f}} = \mathrm{S}^{\mathrm{f}}}$	(8)
$\dfrac{(x \xrightarrow{a} x' \in E), (x \neq s)}{x \xrightarrow{a} x' \in T}$	(9)
$\dfrac{(x = s)}{(x \xrightarrow{f_s(begin)} q) \in T, (q \xrightarrow{f_s(end)} q') \in T}$	(10)
$\dfrac{(x \xrightarrow{a} x' \in E), (x = s)}{(s \xrightarrow{a} x' \in T), (q' \xrightarrow{a} x' \in T)}$	(11)
Interrupt_with(X,Y)Where φ(φ holds in state s)	
$\dfrac{S}{\mathrm{Q} = \mathrm{S} \cup \{\mathrm{q}, \mathrm{q}'\}}$	(12)
$\dfrac{S^{f}}{\mathrm{Q}^{\mathrm{f}} = \mathrm{S}^{\mathrm{f}} \cup \{\mathrm{q}'\}}$	(13)
$\dfrac{(x \xrightarrow{a} x' \in E)}{x \xrightarrow{a} x' \in T}$	(14)
$\dfrac{(x = s)}{(x \xrightarrow{f_s(begin)} q) \in T, (q \xrightarrow{f_s(end)} q') \in T}$	(15)



Use Case X
Basebuilder of the assignment Z:=Include (X,Y)Where P
Basebuilder of the assignment W:=Interrupt_with (X,Y) Where P
Basebuilder of the assignment Y:=Extend_with . (X,Y) Where P
Legend
Initial State
Final State
State Where P holds
(a)
(b)
(c)
(d)

Fig. 4: Examples of Base Builder of an assignment : (a) base use case (b) synthesized base builder with include operator in state x_1 (c) synthesized base builder with Interrupt_with operator in state x_1 (d) synthesized base builder with Extend_with operator in state x_1

4.2.3 Synthesis of Referred Builders

The synthesis of the referred builder is independent of the operator and the states in the join points set. Each referred builder is synthesized from clones of the referred use case using the following rules. Let $A^s_{2_{clone}} = (S_2, s^0_2, S^f_2, L^s_2, E^s_2)$ the FSA of the referred use case clone synthesized from A_2 *with* the renaming function f_s.

The referred builder of A^s_2 with the same renaming f_s is a use case FSA $A^s_2 = (Q, q^0, Q^f, L^s_2 \cup \{f_s(begin), f_s(end)\}, T)$ such that:

$$\frac{S}{(\mathrm{Q} = \mathrm{S} \cup \{\mathrm{q}\})} \qquad (16)$$

$$\frac{S}{(\mathrm{Q}^{\mathrm{f}} = \{\mathrm{q}\})} \qquad (17)$$

$$\frac{s^0}{q \xrightarrow{f_s(begin)} s^0 \in T} \qquad (18)$$

$$\frac{s \xrightarrow{a} s' \in E}{s \xrightarrow{a} s' \in T} \tag{19}$$

$$\frac{s \in S^f}{s \xrightarrow{f_s(end)} q \in T} \tag{20}$$

Rule (16) defines the set of states of the referred builder as the set of states of the referred use case with an additional state q. Rule (17) defines the set of final states of the referred builder. According to Rule (18), a transition is fired from the initial state of the builder to the corresponding state of the initial state of the referred use case. This transition is labeled with f_s*(begin)*. Rule (19) implies that the builder evolves as the referred use case. Finally, Rule (20) reflects that all the final states are transited to the unique final state of A_2^s with the label f_s*(end)*, which is the initial state of the builder. Fig. 5 illustrates an example of a synthesized referred builder using these rules.

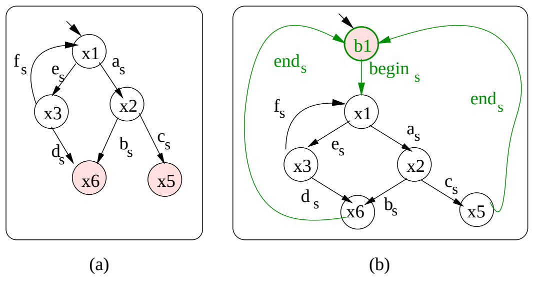


Fig. 5: Example of referred builder (a) referred use case clone with a renaming function f_s(b) its referred builder synthesized using rules (16-20)

4.2.4 Intermediate Use case Generation

When builders are generated, their composition is achieved within their synchronized product on common labels (using Definition 2). During this synchronization, the referred builders will never synchronize since they have different edges labeling. In addition, referred and base builders synchronize only on f_s*(begin)* and f_s*(end)*, $s \in J$. Hence, we verify that $L_1 \cap (\bigcup_{s\in J} L_2^s) = \varnothing$. This verification does not constraint the approach. In fact, if the intersection is not the empty set, a simple renaming for the common labels can be made and then recovered after the synchronization.

The resulting automaton still does not represent the intermediate use case since some of its transitions are labeled by f_s*(begin)* and f_s*(end)*, $s \in J$. These transitions are treated as ε-transition and removed using the ε-transition removal algorithm in [2]. They were needed only for the generation of the synchronized product of the builders reflecting the semantics of the composition operator in the join points. After this step, the synthesized FSA represents the intermediate use case. It is illustrated in step (4) in Fig. 6.

4.2.5 Labeling and Final States Recovery

Let $A_1 = (S_1, s^0_1, S^f_1, L_1, E_1)$ be the base use case and $\{A^s_{2_{clone}}, s \in J\}$ the set of the referred use case clones where J is the set of join points. Let $C = (Q, q^0, Q^f, L_1 \cup (\bigcup_{s \in J} L^s_2), T)$ the generated intermediate use case. We call it intermediate since it still holds the renaming of labels used to generate the different clones of the referred use case. Therefore, we have to restore the original labeling to gain the final use case. For this purpose, we define a renaming function g such that:

$$g : L_1 \cup (\bigcup_{s \in J} L^s_2) \rightarrow L_1 \cup L_2 \ \ where:$$

$$\begin{cases} \forall l \in L_1, g(l) = l \\ \forall l \in L_s, s \in J, g(f_s(l)) = l \end{cases}$$

The label restoration of the intermediate use case results in the new use case as shown in step(5) of Fig. 6. By determining the set of final states, the final use case would be achieved. The set of final states S^f of the newly generated use case $D=(S, s^0, S^f, L_1 \cup L_2, E)$ is defined with respect to the composition operator specified between A_1 and A_2. In the case of `Include` and `Extend_with` composition operators, the set of final states of the new use case represents all the states labeled by one of the final state of the base use case.

$$\frac{((s_1, s_2, ..., s_n) \in S), (s_i \in S^f_1)}{(s_1, s_2, ..., s_n) \in S^f} \tag{21}$$

However, in the case of `Interrupt_with` composition operator, the set of final states of the new use case represents the union of all the states that are labeled by one of the final states of the base use case or the referred use case. This stems from the fact that the `Interrupt_with` operator does not let resumption of the base use case after the execution. Therefore the set of the final states in this case follow the rule (22) as well as the rule (21):

$$\frac{((s_1, s_2, ..., s_n) \in S), (s_i \in S^f_2)}{(s_1, s_2, ..., s_n) \in S^f} \tag{22}$$

It is important to mention that unlike the approach in [3] , our approach does not introduce any non-determinism. In fact, if the use cases specified are deterministic, the generated use cases from assignments would be also deterministic. An example of the overall process of the composition is shown in Fig. 6.

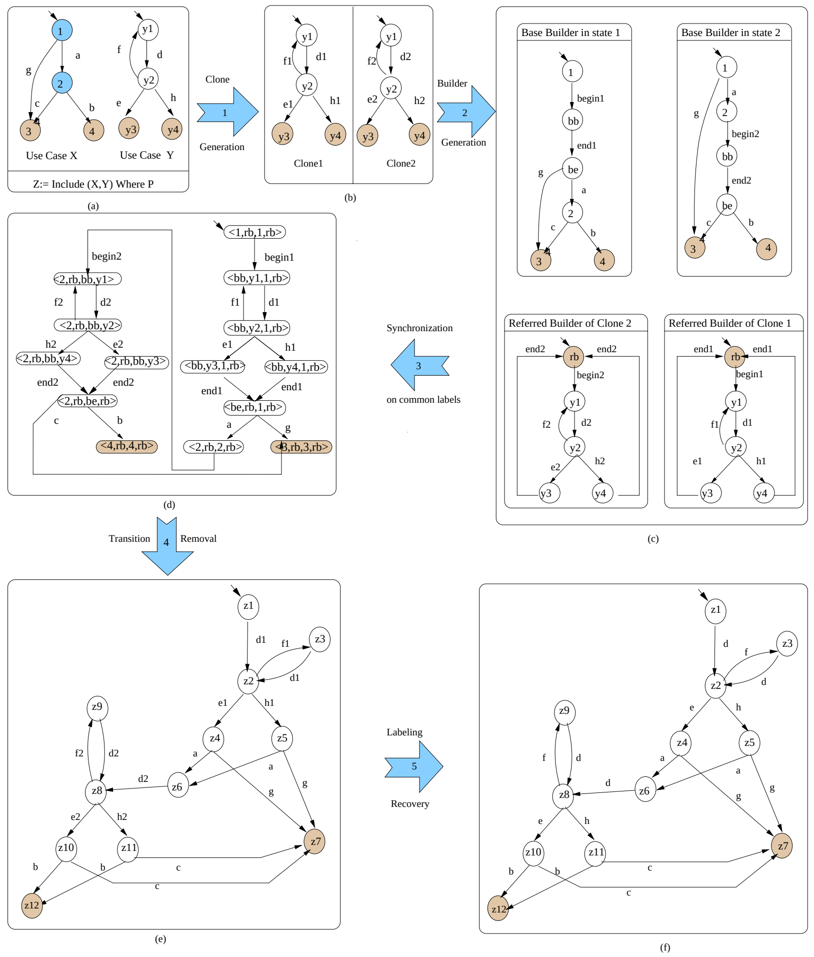


Fig. 6: Example of Use Case Composition using Assignment Expression: (a) original specification: use case X and Y and the assignment Z (b) clones of the referred use case (c) builders of the use case X in state 1 and state 2 as well as the referred builders of the clones of Y (d) Synchronized product of base and referred builders (e) the intermediate use case (f) the generated use case Z

5 Application on Distributed use cases

Distributed use cases are those where the communication between the different entities is described. In order to show the applicability of our approach on the distributed systems, we choose the specification of a distributed Invoice Ordering System.

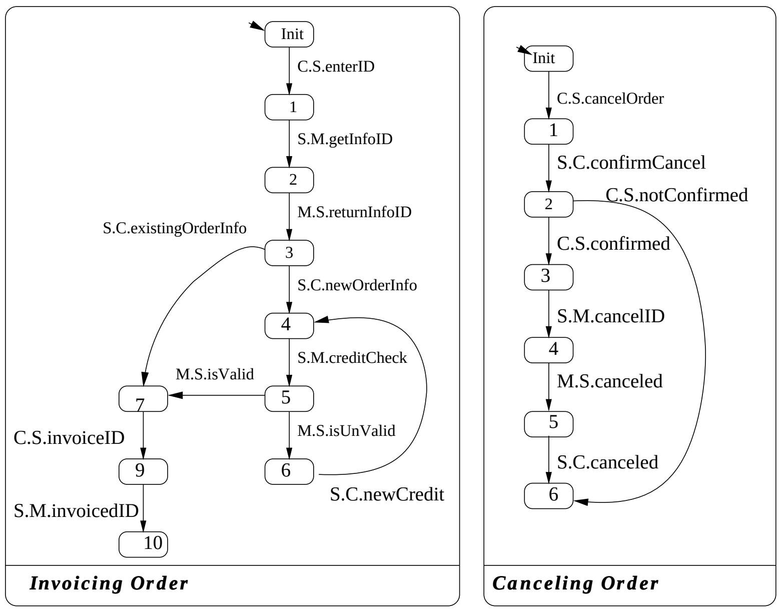


Fig. 7: Invoice and Cancel order use cases

In order to let our use case model handle the description of distributed use cases, we represent the labeling of the FSA in the form of (O_1.O_2.m) where O_1 and O_2 are the communicating objects, and m is the message sent from the object O_1 to the object O_2 [4]. We present in Fig. 7 two use cases of the invoice ordering system specification. Three objects are communicating in the system: the customer (C), the system (S), and the resource manager (M). Let's build a new use case, *Ordering*, where it shows that the costumer is authorized to cancel its ordering if the order is not yet invoiced. The assignment is:

```
Ordering: = Interrupt_with (Invoicing, Canceling) Where
       (AG(orderID)∧ AG((!invoiced)U (invoiced))
```

The CTL property states that the customer may cancel his order from the time of receiving the confirmation of his order ID (new or existing) and before invoicing his order. According to this assignment, the set of states that verify the property is {3,4,5,6,7}. The use case *Canceling order* will be composed with the `Interrupt_With` semantics in those states. We note that after the composition of use cases, it is possible to decompose the obtained FSA to communicating FSAs per object. This could be done with the projection of the behavior on the objects, as presented in our previous work [4].

6 Related Work and Discussions

Many approaches have been developed to synthesize state-based models from a set of use cases [5-11]. State-based models are basically needed to verify and validate the user requirements in order to detect design problems as soon as possible. In this paper we tackled the issue of automatic generation of system automaton based on use case composition through assignment evaluation.

The emerged notations to specify use cases have different degrees of expressiveness and formality. Glinz [12] uses statecharts to model scenarios. The integration of scenarios is performed 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 [13] introduces a new kind of chart and notation to model dependencies among scenarios. The advantage of this approach is the fact of capturing clearly these inter-scenarios dependencies. Yet, this work is presenting a notation rather than a methodology that can clarify the dependencies between different scenarios. Bordeleau *et al.* [14] have proposed integration patterns for scenario dependencies. UCMs are used to detect dependencies between scenarios. A state-based specification per use case is generated for each component and integrated to reflect the scenarios dependencies. The whole process is done manually and relies on the creativity of the analyst to connect together the different statecharts in the right way. Araujo *et al.* [15] focuses on representing aspects during the use case modeling. They propose 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 through an interaction pattern and defined in term of roles. In our case, we propose composition operators to generate new use cases that integrate the behaviors of the original ones.

During the composition of use cases into transition-based system, the challenge is to identify states at the scenario level that serve as join points between use cases. There are two kinds of state characterization: trace-based [5-7, 16], and variable (or label) state-based characterization [8, 9, 17]. In this paper, we propose to detect these states using a model checking approach. The state where the composition has to be made verifies a certain property of the use case. This helps considerably the analyst since he has no more the arduous task to detect the right state.

Our approach differs substantially from the earlier presented work in some points. During the process of generating the specification, the analyst has the opportunity to define assignments in an incremental manner. Hence the order in which these assignments are specified has a direct impact on the resulting use cases. In fact, the states that will be generated from the model checking system will differ with different orders in presenting the assignments. Consequently, having different combinations of defining assignments and then choosing the proper order may ease the process of obtaining the expected behavior. In addition, we kept the model as rich as possible by having use cases described as FSAs without complicating the composition procedure. Having well-established synthesis rules for each composition operator and a composition based on the well known concept of synchronized product makes the composition automated, formal and straightforward.

7 Conclusion

In this paper, we presented an approach for composing use cases based on the notion of assignments. Each assignment includes a base use case, a referred use case, a composition operator, and a CTL property which is used to identify the states on which the composition will be done, called joint points. The CTL property and the base use case are sent to a model checking tool in order to determine the joint points, where to perform the composition. The composition approach consists of three steps. First, clones of the referred use case are generated using a renaming function. Then, proper builders that reflect the semantics of the composition operator and the join pints are synthesized and their synchronized product on common labels is generated. Finally the obtained automaton is processed through a relabeling function in order to recover the original labeling. The obtained FSA represents the behavior of the base and the referred use cases, merged on the states which hold the specified property with respect to the semantics of the composition operator.

Our approach is fully automated because of the synthesis rules for constructing builders and the synchronization mechanism used for composition. It also has the advantage of providing a helpful support for the analyst, especially when join points are not clearly evident, which may be the case proceeding with the composition. In fact, the size of the use case automaton grows significantly after the composition, adding to the complexity of specifying join points manually. Furthermore, an optional validation by the analyst after the selection of the join points may be envisaged. The expansion and enrichment of the model as well as the composition approach is seen as a part of future work. A tool supporting and visualizing the composition approach is under construction.

References

[1] E. M. Clarke, J. O. Grumberg, and D. A. Peled, *Model Checking*: MIT Press, 1999.

[2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* second edition ed: Addison Wesley 2000.

[3] R. Mizouni, A. Salah, R. Dssouli, and S. Kolahi, "Role of Variables and Interactions in Use Case Composition," presented at New Technologies for Distributed Systems (NOTERE'06), Toulouse, France, 2006.

[4] A. Salah, R. Mizouni, R. Dssouli, and B. Parreaux, "Formal Composition of Distributed Scenario," presented at FORTE : International Conference on Formal Techniques for Networked and Distributed Systems, Spain, 2004.

[5] D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," *Int. J. of Foundations of Computer Science*, vol. 13, pp. 5-51, 2002.

[6] K. Koskimies and E. Mäkinen, "Automatic Synthesis of State Machines from Trace Diagrams," *Software-Practice and Experience*, vol. 24, pp. 643-658, 1994.

[7] E. Mäkinen and T. Systä, "MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML," presented at ICSE 2001, Toronto, Canada, 2001.

[8] R. Dssouli, S. Some, J. Vaucher, and A. Salah, "Service creation environment based on scenarios," *Information and Software Technology*, vol. 41, pp. 697-713, 1999.

[9] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transactions on Software Engineering*, vol. 29, pp. 99-115, 2003.

[10] J. S. Jon Whittle, "Generating statechart designs from scenarios.," presented at the 22nd International Conference on Software Engineering, 2000.

[11] D. Amyot, W. D. Cho, X. He, and Y. He, "Generating Scenarios from Use Case Map Specifications," presented at Third International Conference on Quality Software (QSIC'03), Dallas, November 2003.

[12] M. Glinz, "An integrated formal model of scenarios based on statecharts," presented at Proceedings of the~Fifth~European Software Engineering Conference, 1995.

[13] J. Ryser and M. Glinz, "Dependency Charts as a Means to Model Inter-Scenario
Dependencies," presented at In G. Engels, A. Oberweis and A. Zündorf (eds.): Modellierung 2001. GI-Workshop, volume P-1, Bad Lippspringe, Germany, 2001.

[14] F. Bordeleau and J. P. Corriveau, "On the Importance of Inter-Scenario Relationships in Hierarchical State Machine Design," presented at In Proceedings of Fundamental Approaches to Software Engineering (FASE'2001), held as part of the Joint European Conferences on Theory and Practice of Software ETAPS'2001., Genova, Italy, 2001.

[15] J. W. J Araújo, D-K Kim, "Modeling and Composing Scenario-Based Requirements with Aspects " presented at the 12th IEEE International Requirements Engineering Conference (RE'04), Kyoto, Japan, 2004.

[16] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," presented at Distributed and Parallel Embedded Systems, 1998.

[17] A. Salah, R. Dssouli, and G. Lapalme, "Compiling real-time scenarios into a Timed Automaton," presented at FORTE : International Conference on Formal Techniques for Networked and Distributed Systems, 2001.