Revisiting statechart synthesis with an algebraic approach

Tewfic Ziadi, Loïc Hélouët, Jean-Marc Jézéquel

Revisiting Statechart Synthesis with an Algebraic Approach

Tewfik Ziadi, Loïc Hélouët, Jean-Marc Jézéquel
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{tewfik.ziadi, loic.helouet, jezequel}@irisa.fr

Abstract

The idea of synthesizing statecharts out of a collection of scenarios has received a lot of attention in recent years. However due to the poor expressive power of first generation scenario languages, including UML1.x sequence diagrams, the proposed solutions often use ad hoc tricks and suffer from many shortcomings. The recent adoption in UML2.0 of a richer scenario language, including interesting composition operators, now makes it possible to revisit the problem of statechart synthesis with a radically new approach. Inspired by the way UML2.0 sequence diagrams can be algebraically composed, we first define an algebraic framework for composing statecharts. Then we show how to leverage the algebraic structure of UML2.0 sequence diagrams to get a direct algorithm for synthesizing a composition of statecharts out of them. The synthesized statecharts exhibit interesting properties that make them particularly useful as a basis for the detailed design process. Beyond offering a systematic and semantically well founded method, another interest of our approach lies in its flexibility: the modification or replacement of a given scenario has a limited impact on the synthesis process, thus fostering a better traceability between the requirements and the detailed design.[1]

1. Introduction

Scenario languages such as UML Sequence Diagrams (SD) are often used to capture behavioral requirements of a system. Requirements may contain usual behaviors expected from the system as well as exceptional cases. Scenarios represent a global view of cooperations inside a system. They are close to human understanding and usually remain rather abstract and unprecise. While it seems illusory to try to define a system by trying to design "all its scenarios", the idea of synthesizing statecharts out of a collection of scenarios has received a lot of attention in recent years. This is probably because designing a system behavior directly with statecharts is not a intuitive process, as the notion of state is often not natural in early stages of development. As pointed out by [7], a sequence diagram is an inter-object view of a system, i.e. an history implying a cooperation of several objects to realize a functionality, while a statechart can be considered as an intra-object description, that includes several functionalities and is closer to an implementation.

Due to the poor expressive power of first generation scenario languages, including UML1.x sequence diagrams, the proposed solutions for statechart synthesis [21, 11, 16, 10] often use ad hoc tricks and suffer from many shortcomings. The recent adoption in UML2.0 of a richer scenario language, including interesting composition operators, now makes it possible to revisit the problem of statechart synthesis with a radically new approach.

Inspired by the way UML2.0 sequence diagrams can be algebraically composed, we first define an algebraic framework for composing statecharts. Then we show how to leverage the algebraic structure of UML2.0 sequence diagrams to get a direct algorithm for synthesizing statecharts: we propose to transform scenarios given as a composition of sequence diagrams (as defined in UML2.0) into a composition of state machines. Beyond offering a systematic and semantically well founded method, another interest of our approach lies in its flexibility: the modification or replacement of a given scenario has a limited impact on the synthesis process, thus fostering a better traceability between the requirements and the detailed design.

This paper is organized as follows: Section 2 introduces the main concepts and notations used throughout the paper through the well known ATM (Automatic Teller Machine) example [20, 21]. It goes on by introducing our algebraic framework for composing statecharts. Section 3 describes our synthesis algorithm and illustrates it on the ATM example. Sec-

[1]This work has been partially supported by the FAMILIES European project. Eureka $\sum$! 2023 Program, ITEA project ip 02009.

tion 4 discusses the role and limitations of synthesis in a development process, including the precise semantic relationship existing between the scenarios and the synthesized statecharts. Section 5 compares our approach with related works.

2. Scenarios and statecharts

Scenarios are used to define systems behavioral requirements. They are close to users understanding and they are often used to refine use cases and provide an abstract view of a system. Several notations have been proposed, among which UML sequence diagrams[5], message sequence charts(MSCs) [9], and live sequence charts [3]. In this paper we focus on scenarios represented as UML2.0 sequence diagrams (SDs). Scenarios are not the only way to capture behaviors of a system, and a formalism like statecharts [6] can also be used. However, even if both views depict behavioral aspects of a system, they have a very different nature. While scenarios capture interactions between a set of objects, statecharts, represent the internal behavior of a single object. As underlined in [7], scenarios are more an inter-object view of system behaviors while statecharts are an intra-object view of the same system.

An important question concerning synthesis is the relationship between the initial scenario model and the synthesized state machines. Should the synthesized behaviors be exactly the same, contain or be contained in the original behaviors given by scenarios ? Synthesizing objects that do not even fulfill initial requirements does not really make sense, so the last option can be forgotten. Because of the incompleteness of typical scenarios, statechart synthesis should be more considered as a step towards an implementation rather that as a definitive bridge from user requirements to code. Hence, the most sensible relation required between inter and intra views is that requirement should be at least included in the synthesized objects behaviors. Section 4 will show that behavior equality or inclusion is only possible under certain assumptions about communication between state machines. In addition to this, requiring equivalence between inter and intra views behaviors is only possible when reducing the expressive power of the scenario language.

The approach proposed hereafter revisits the problem of statecharts synthesis with an algebraic approach allowing to switch from an algebraic composition of SD to an algebraic composition of statecharts. We have assumed an asynchronous communication model between communicating state machines, which allows systematically the inclusion of scenarios in synthesized behaviors. In the rest of this section, we first present UML2.0 SDs and their algebraic composition, and then introduce an algebraic framework for statecharts composition.

2.1. UML2.0 Sequence Diagrams

UML2.0 [5] Sequence diagrams greatly enhance the previous versions of scenarios proposed in UML1.x. Basic Sequence diagrams describe a finite number of interactions between a set of objects. They are now considered as collections of events (instead of ordered collections of messages in UML1.x), which introduces concurrency and asynchronism, and allows the definition of more complex behaviors. In addition to this, sequence diagrams can now be composed by means of operators to obtain more complex interactions.

Figure 1 shows five basic SDs defining possible scenarios for a well known example, the ATM (Automatic Teller Machine). We only work on a part of the ATM behaviors defining the introduction of a card, its removal, and the user identification. A UML2.0 SD is represented by a rectangular frame labeled by the keyword **sd** followed by the name of the SD. The sequence diagram `EnterPassword` of Figure 1 describes the interactions of four objects User, ATM, Consortium and Bank. The vertical lines represent life-lines for the given objects. Interactions between objects are shown as horizontal arrows called messages (like "enterPassword"). Each message is defined by two events: message emission and message reception, which induces an ordering between emission and reception. Events situated on the same lifeline are ordered from top to down.

Definition 1 *A* basic Sequence diagram *is a tuple* $(E, \leq, \alpha, \phi, A, I)$ *where* E *is a set of events,* $\leq$ *is a partial ordering imposed by lifelines and messages,* A *is a set of actions (message emissions and receptions),* I *is a set of objects participating to the interaction, and* α *and* ϕ *are mappings associating respectively an action name and a location (i.e an object affected by the event) to an event.*

Sequence diagram `UserCancel` in Figure 1 shows the interactions between an User and the ATM when a transaction is cancelled. Note that interactions are not mandatorily synchronous, as in UML1.x. Hence, messages `EjectCard` can be sent before reception of message `cancelledMessage`.

Basic SDs only represent finite behaviors without branching (when executing a Sequence diagram, the only branching is due to interleaving of concurrent events), but can be composed to obtain more complete descriptions. UML2.0 basic SDs can be composed in a composite SD called *combined interaction* using a set of operators called *interaction operators*. The three fundamental operators are: `seq`, `alt`, and `loop`. The `seq`

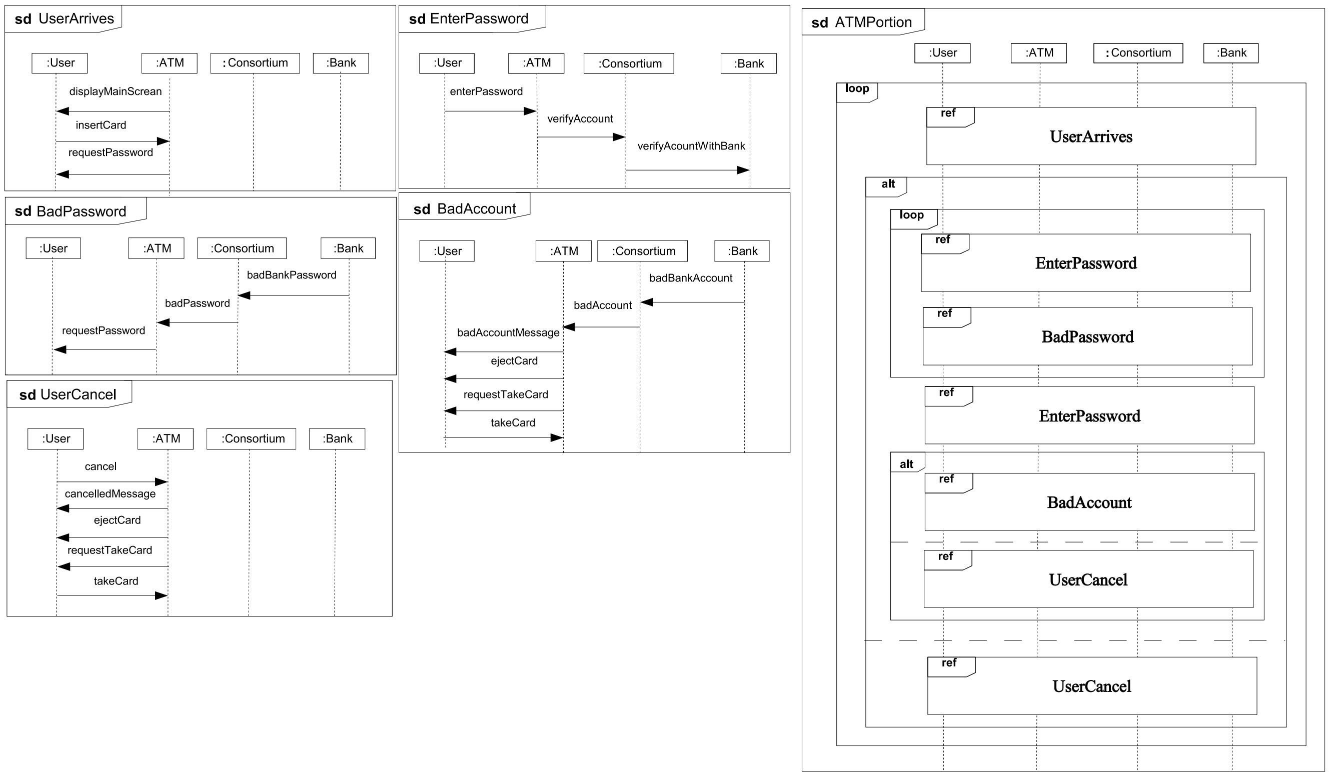

Figure 1. Sequence diagrams for the ATM example

operator specifies a weak sequence between the behaviors of two operand SDs (all events in the first operand situated on an object o must be executed before events of the second operand situated on the same object). The `alt` operator defines a choice between a set of interaction operands. The `loop` operator specifies an iteration of an interaction.

For all these operators, each operand is either a basic or a combined SD. The combined SD `ATMPortion` in Figure 1 composes five basic SDs using operators. References to SD are described by a rectangular frame labeled by the keyword **ref** in the upper left corner and containing the name of the referred SD. The composition operators are described by rectangles which left corner is labeled by an operator (`alt`, `seq`, `loop`). Operands for sequence and alternative are separated by dashed horizontal lines. Sequential composition can be also implicitly given by the relative order of two frames in a diagram. For example, in the SD `ATMPortion` the basic SD `EnterPassword` is referenced before the SD `BadPassword`. This is equivalent to the expression `EnterPassword seq BadPassword`. Composition operators can be seen as defining regular expressions on a set of sequence diagrams, that will be called references expressions for SDs.

Definition 2 *A* references expression *for sequence diagrams (noted RESD hereafter) is an expression of the form:*
`E ::= SD | (E` **alt** `E) | (E` **seq** `E) |` **loop** `( E )`
where SD is a reference to a basic sequence diagram and `seq`*,* `alt` *and* `loop` *are the SD operators mentioned above.*

Let us consider the SD `ATMPortion` of Figure 1. This SD can be represented by the following expression:

```
E = loop( UserArrives seq (loop(
EnterPassword seq BadPassword ) seq
(EnterPassword seq          (BadAccount alt
UserCancel)) alt UserCancel))
```

2.2. Algebraic framework for statecharts

We propose to define an algebraic framework for statechart composition in a similar way. We formalize three operators allowing sequential composition, alternative and iteration of statecharts. We use reference expressions for statecharts as an algebraic specification of statechart composition. So far, we do not consider concurrency along an object's lifeline in a SD. We will not need high-level constructs in statecharts such as hierarchy and concurrent states. We will only use *flat* statecharts.

Definition 3 *A* flat *statechart is a 6-tuple* $\langle S, s_0, E, A, \delta, J\rangle$ *where* S *is a set of states,* s_0 *is the initial state,* E *is a set of events,* A *is a set of actions,* $\delta \subseteq S \times E \times A \times S$ *is the transition relation.* $J \subseteq S$ *is a set of junction states.*

Junction states are close to the usual notion of final states in classical automatas, but will have an additional role during statechart composition (they will be a kind of "merging states" for some operators). Transitions can be either:

- $(s, \emptyset, a, s')$, which corresponds to message emission. Transitions of this kind will be denoted by an arrow from the starting state to the target state, and labeled by $/a$.
- $(s, e, \emptyset, s')$, which corresponds to message receptions. Transitions of this kind will be denoted by an arrow from the origin state to the target state, and labeled by e.

Note that we have not adopted the usual event/reaction notation for transitions, as we think that message emissions can result from internal choices that are not represented in an interaction, and can not be systematically depicted as reactions to a message reception. However, compacting statecharts transitions to obtain transitions of the kind $reception/emission_1, emission_2, ...$ is surely possible in many cases.

Figure 2 shows examples of flat statecharts, in which junction states are represented by double circled states. $ST_\emptyset$ refers to an empty statechart, containing a single state which is at the same time an initial and a junction state (see statechart $ST_\emptyset$ in Figure 2).

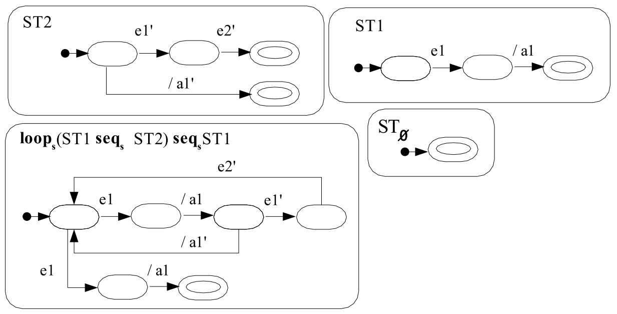

Figure 2. Flat statecharts

2.3. Statecharts operators

We formalize three statechart operators: $\texttt{seq}_s$, $\texttt{loop}_s$ and $\texttt{alt}_s$ respectively for the sequential composition, the iteration and the alternative composition of statecharts. Junction states that have been introduced previously will be necessary to formalize these operators. A statechart ST is a loop if the initial state is a junction state, and if it is not an empty statechart (i.e $s_0 \in J \wedge ST \neq ST_\emptyset$). Equality between statecharts is defined as isomorphism between their definition.

Let $ST1 = \langle S^1, s_0^1, E^1, A^1, \delta^1, J^1\rangle$ and $ST2 = \langle S^2, s_0^2, E^2, A^2, \delta^2, J^2\rangle$ be two flat statecharts.

Sequence ($\texttt{seq}_s$). The sequential composition of two statecharts is a statechart that describes the behavior of the first operand *followed* by the behavior of the second one. $ST1\ seq_s\ ST2 = \langle S, s_0, E, \mathcal{A}, \delta, J\rangle$, where:

- The initial state of $ST1\ seq_s\ ST2$ is the initial state of the first statechart if it is not empty and of the second one otherwise.

$$s_0 = \begin{cases} s_0^1 \text{ if } ST1 \neq ST_\emptyset \\ s_0^2 \text{ otherwise} \end{cases}$$

- $S = \begin{cases} S^1 \cup S^2 - \{s_0^2\} \quad \text{if } (s_0^2 \notin J^2 \vee ST2 = ST_\emptyset) \\ S^2 \quad \text{if } ST1 = ST_\emptyset \\ S^1 \cup S^2 \text{ otherwise} \end{cases}$

- $E = E^1 \cup E^2$; $A = A^1 \cup A^2$: events and actions of $ST1\ seq_s\ ST2$ are the union of those in the two operands.

- Sequential composition of two statecharts preserves all transitions of its operands, except transitions from the initial state of $ST2$ when $ST2$ is not a loop. For the concatenation of two statecharts, new transitions are added from each junction state of the first statechart to all successors of the initial state of the second one. This is defined as: $\delta = \delta^1 \cup (\delta_2 \cap S \times E \times A \times S) \cup \{(j, e, a, s) \in J^1 \times E^2 \times A^2 \times S^2 | (s_0^2, e, a, s) \in \delta^2)\}$

- $J = \begin{cases} J^2 \cup J^1 \quad \text{if } s_0^2 \in J^2 \\ J^2 \text{ otherwise} \end{cases}$

$ST_\emptyset$ is a neutral element for sequential composition, i.e. for any statechart ST, $ST\ seq_s\ ST_\emptyset = ST_\emptyset\ seq_s\ ST = ST$.

Loop ($\texttt{loop}_s$). This operator defines the *iteration* of a statechart. $loop_s(ST1) = \langle S, s_0, E, \mathcal{A}, \delta, J\rangle$, where:

- the initial state of the iterated statechart remains unchanged, i.e. $s_0 = s_0^1$. S contains all states excepting junction states, i.e. $S = (S^1 - J^1) \cup \{s_0^1\}$.

- $E = E^1$; $A = A^1$.
- Iteration adds transitions from predecessors of each junction state of the statechart to the initial state, and removes transitions to junction states. This is defined as: $\delta = (\delta^1 \cap S \times E \times A \times S) \cup \{(s, e, a, s_0^1) \mid (s, e, a, j) \in \delta^1\}$
- the resulting statechart only contains the initial state as junction state. i.e. $J = \{s_0^1\}$.

The iteration of the empty statechart is the empty statechart itself i.e. `loop`$_s$ `(ST`$_\emptyset$`) = ST`$_\emptyset$.

Alternative (`alt`$_s$). The statechart resulting from the alternative composition describes a *choice* between the behaviors of its operands. $ST1\ alt_s\ ST2 = \langle S, s_0, E, A, \delta, J\rangle$, where:

- $$s_0 = \begin{cases} \text{a new state } s \text{ if ST1 and ST2 are loops,} \\ i.e.\ (s_0^1 \in J^1 \wedge s_0^2 \in J^2 \wedge ST1 \neq ST_\emptyset \wedge \\ \quad ST2 \neq ST_\emptyset) \\ \\ s_0^2 \text{ if only ST1 is a loop or empty,} \\ i.e.\ (s_0^1 \in J^1 \vee ST1 = ST_\emptyset) \wedge s_0^2 \notin J^2 \\ \\ s_0^1 \text{ otherwise} \end{cases}$$
  Note that we keep s_0^1 as initial state by default, but that we obtain a similar result when keeping s_0^2.
- $$S = \begin{cases} S^1 \quad \text{if } (ST2 = ST_\emptyset \wedge ST1 \neq ST_\emptyset) \\ S^2 \quad \text{if } (ST1 = ST_\emptyset \wedge ST2 \neq ST_\emptyset) \\ \{s_0\} \quad \text{if } (ST1 = ST_\emptyset \wedge ST2 = ST_\emptyset) \\ \\ S^1 \cup S^2 \cup \{s\} \text{ if } (s_0^1 \in J^1 \wedge s_0^2 \in J^2 \wedge \\ \qquad ST1 \neq ST_\emptyset \wedge ST2 \neq ST_\emptyset) \\ \\ S^1 \cup S^2 - \{s_0^2\} \quad \text{if } s_0^1 \notin J^1 \wedge s_0^2 \notin J^2 \\ S^1 \cup S^2 \quad \text{otherwise} \end{cases}$$
- $E = E^1 \cup E^2$; $A = A^1 \cup A^2$.
- To specify a choice between the behaviors of the two statecharts, new transitions are added from the new initial state of to all successors of the initial states of the operands. This is defined as:
  $$\begin{aligned} \delta = \ & (\delta^1 \cap S \times E \times A \times S) \\ & \cup(\delta^2 \cap S \times E \times A \times S) \\ & \cup\{(s_0, e, a, s) \mid (s_0^1, e, a, s) \in \delta^1 \\ & \vee(s_0^2, e, a, s) \in \delta^2\} \end{aligned}$$
- junction states are the union of junction states of operands i.e. $J = (J^1 \cup J^2) \cap S$.

$ST_\emptyset$ is a neutral element for choice, i.e $ST\ alt_s\ ST_\emptyset = ST_\emptyset\ alt_s\ ST = ST$.

As for sequence diagrams, we describe algebraically statecharts composition as reference expressions.

Definition 4 *A* Reference expression *for statecharts (noted REST hereafter) is an expression of the form:*
`E ::= ST | E seq`$_s$ `E | E alt`$_s$ `E | loop`$_s$ `(E)`

The expression $loop_s(ST1\ alt_s\ ST2)$ is an example of REST. The flat statechart associated to this expression is obtained by applying alternative to $ST1$ and $ST2$ and then the loop operator on the result. Note that the statecharts obtained after composition are not necessarily deterministic (see for example, the statechart obtained from the expression $loop_s(ST1 seq_s ST2) seq_s ST1$ in Figure 2). However, they can be transformed into deterministic automata using standard algorithms once the synthesis process is accomplished.

3. Generating statecharts

This section proposes an algorithm generating flat statecharts from UML2.0 SDs. First, we show how basic statecharts are generated from basic SDs. Then, we define the generation of statecharts from combined SDs as a mapping from RESD to REST.

3.1. Basic Sequence Diagrams

The generation of statechart for a given object from a basic SD is based on the projection of the SD events on the object's life-line. Remember that events situated on the same lifeline are totally ordered.

Definition 5 *The* projection $\pi_O(S)$ *of a SD S on an object O is the restriction of the order $\leq$ to events situated on O's lifeline. As this restriction is a total order, we will consider the projection as the word $\pi_O = e_1.e_2 \ldots e_n$ such that $\{e_1, \ldots e_n\} = \phi^{-1}(O)$, and $e_1 < e_2 < \ldots e_n$.*

Let us denote by $!m$ the sending of message m and by $?m$ the corresponding reception. The word $!displayMainScrean.?insertCard.!requestPassword$ is the projection of the SD `UserArrives` of Figure 1 on the "ATM" lifeline. Receptions in the SD become events in the statechart and emissions become actions. For a transition associated to a reception, the action part will be empty, and for transitions associated to actions, the event part will be empty.

The following algorithm shows how to generate a flat statechart for a given object O from a basic SD

S. Clearly, statecharts generated will be sequences of states, and will contain a single junction state, that corresponds to the state reached when all events situated on an object lifeline have been executed. Note that when an object does not participate in an interaction, the projection of a SD on this object's lifeline is the empty word, noted ϵ. For this specific case, the generated statechart is $\mathtt{ST}_\emptyset$.

algorithm: $P(S, O)$

```
  Input :   A basic SD $S$, an object $O$
  Output : A statechart $ST_O = (S, s_0, E, A, \delta, J)$
  Create the initial state $s_0$
  $currentState := s_0$
  $E := \emptyset$; $A := \emptyset$; $S := \{s_0\}$; $J = \emptyset$; $\delta = \emptyset$
  $ProjectedEvents := \pi_O(S)$
  if $ProjectedEvents$ is empty then
    return($ST_\emptyset$)
  else
    for $i = 1$ to $|ProjectedEvents|$ do
      $e_i := ProjectedEvents[i]$
      Create a new state $s$; $S = S \cup \{s\}$
      if $e_i$ is a receiving event then
        $E := E \cup \{e_i\}$
        Tr := (currentState, $e_i$, $\emptyset$, $s$)
        $\delta := \delta \cup \{Tr\}$
      else
        if $e_i$ is a sending event then
          $A := A \cup \{e_i\}$
          Tr := (currentState, $\emptyset$, $e_i$, $s$)
          $\delta := \delta \cup \{Tr\}$
        end if
      end if
      currentState := $s$
    end for
    $J = currentState$
    return($ST_O$)
  end if
```

Figure 3 shows the flat statecharts generated from the five basic SDs for the "ATM" object.

3.2. Combined Sequence Diagrams

After building a collection of basic statecharts through projections of basic SDs, the extension of the method to SD reference expressions seems quite immediate. Let E be a RESD depicting the interactions of a set of objects $\mathcal{O} = \{O_1, \ldots O_k\}$. For each object O_i, a REST E_i is constructed by replacing in the RESD `seq`, `alt`, and `loop` respectively by statecharts operators $\mathtt{seq}_s$, $\mathtt{alt}_s$, and $\mathtt{loop}_s$, and each reference to a SD S by the statechart $P(S, O_i)$. From the set of REST $\{E'_1, \ldots, E'_k\}$ obtained, k statecharts can be built using statechart composition operators.

Let us apply this construction method to the combined SD `ATMPortion` Figure 1. The "ATM" 's REST is:

$\mathtt{E}_{ATM}$ = $\mathtt{loop}_s$(P(UserArrives, ATM) $\mathtt{seq}_s$ ($\mathtt{loop}_s$(P(EnterPassword, ATM) $\mathtt{seq}_s$ P(BadPassword, ATM)) $\mathtt{seq}_s$ (P(EnterPassword, ATM) $\mathtt{seq}_s$ (P(BadAccount, ATM) $\mathtt{alt}_s$ P(UserCancel, ATM))) $\mathtt{alt}_s$ P(UserCancel,ATM)))

The synthesized statecharts from algebraic expressions are not necessarily minimal. However, smaller statecharts can be obtained by determinization. Figure 4 shows the determinized "ATM" statechart obtained from this expression. Note that since a specific object may not participate to interactions in one or more basic SDs, its REST can refer several times to the empty statechart $\mathtt{ST}_\emptyset$. This REST can be reduced knowing that the empty statechart is a neutral element for the sequential composition and for the alternative, and idempotent for the loop.

4. Discussion

4.1. Coherence between inter-object and intra-object views

Defining statecharts generation from combined SDs as a mapping from RESDs to RESTs gives a certain flexibility to the synthesis process. After a modification of the RESD (adding or removing a SD for example) a part of the previous synthesis result can be reused. However, this simple and immediate synthesis method produces state machines whose behavior does not necessarily exactly match the initial scenarios.

As already mentioned, synthesis must preserve a certain coherence between the inter-object view given by scenarios, and the composition of intra-object views given by statecharts. Within this context, the way objects are supposed to communicate is not innocent. As shown in [4], some communication models do not allow the implementation of even very simple sequence diagrams. To illustrate our remark, let us consider three communication models for statecharts composition: broadcast, synchronous communications, and asynchronous communications with buffers managed by event dispatchers in a SDL-like style. Let us consider the sequence diagram of Figure 5, and the statecharts obtained. If broadcast communication is assumed between state machines, message b can be broadcast before message a, and as $O1$ needs to receive b before sending c, $O1$ and $O2$ will be deadlocked. This

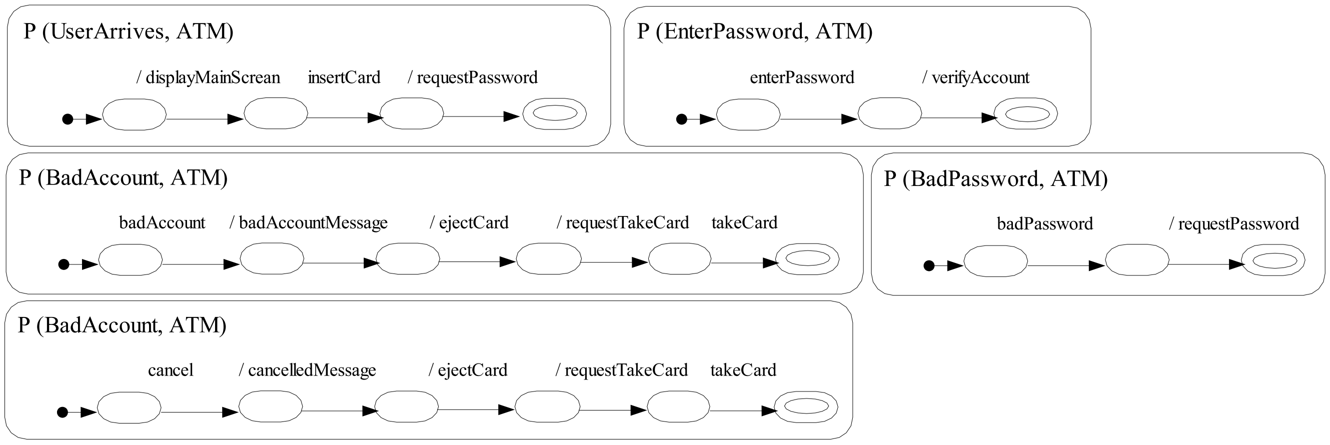

Figure 3. ATM basic statecharts

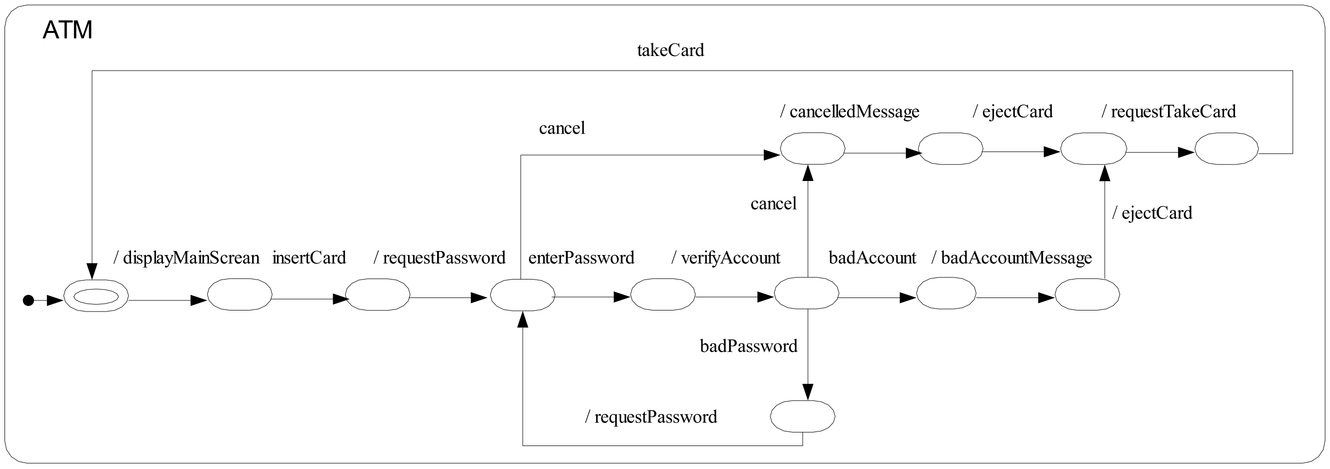

Figure 4. Full statechart for the ATM obtained from SDs of Figure 1

situation does not appear with synchronous or asynchronous communication. It clearly shows that some sequence diagrams cannot be implemented with broadcast communication.

Now, let us consider the example of Figure 6, and the corresponding statecharts synthesized in Figure 7. If object $O1$ sends message a, nothing prevents object $O4$ from sending message d. This leads to an unavoidable deadlock, both in a framework with synchronous and asynchronous communications. However, in SD $Deadlock$, behaviors in $SD1$ and $SD2$ were supposed to be exclusive. Clearly, synthesized machines allow new behaviors. In fact, the choice of a too restrictive communication model (such as broadcast) makes synthesis impossible in some cases, while more permissive communications may produce unexpected behaviors.

Hence, the choice of a communication model is very important for synthesis and has a consequence on the relation between initial requirements and behaviors allowed by generated state machines. Furthermore, the relation between initial requirements and generated statecharts must be the same for all sequence diagrams in order to allow a systematic use of synthesis in a development process. If a communication model only allows the implementation of a subset of the requirements (the behaviors of state machines is systematically included in the behaviors of scenarios), then it may be adequate for some verification tasks, but not really as a step towards an implementation. If the set of generated behaviors **is included** in the requirements for some sequence diagrams, and **contains** the requirements for some others, then statechart synthesis is not really usable in the development process (not as a model refinement step nor for verification purposes). With respect to this remark, assuming broadcast communication for statecharts within the synthesis context is surely a bad choice. A possible approach to deal with these problems is to constrain the use of scenarios in order to ensure that the synthesis process produces state machines with **exactly** the same behaviors that were expressed in the requirements. We do not believe that this approach is realistic in many cases, as scenarios were more designed to express sample behaviors than for exhaustively specifying a system. Reducing the expressivity of scenarios would result in a poor language that would only allow direct implementation in trivial cases. One good compromise is to keep the expressiveness of scenarios, while remaining aware of the gap that still exists between inter and intra views of the system.

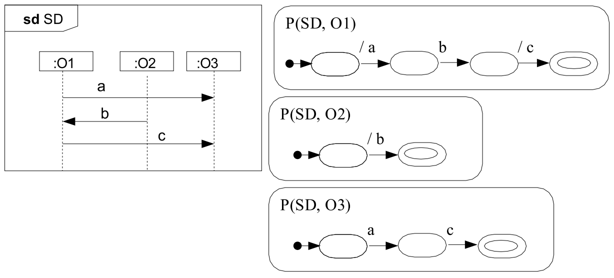

Figure 5. Example for broadcast

4.2. Conditions for behavior inclusion in UML2.0

For a framework such as UML2.0 with asynchronous communications between statecharts, the relation between scenarios and the corresponding state machines synthesized is clearly established. Let us call $T(SD)$ the set of runs depicted by a sequence diagram, and $T(ST)$ the set possible runs defined by a statechart. For a given set of statecharts $\{ST_i\}$, $i \in 1..K$, let us call $\underset{i \in 1..K}{||} ST_i$ the parallel composition of state machines with an appropriate communication mechanism. Let us also assume a very permissive event dispatcher mechanism that associates a fifo buffer to each pair of objects in the system, and can consume the first message needed in this buffer without deleting preceding messages (hence allowing some limited message crossing). Within this framework, it has been proved [8] that $T(SD)) \subseteq T(\underset{o \in O}{||} P(SD, o))$. Having behavior inclusion instead of equality has several consequences on the role that statechart synthesis may play in the design process.

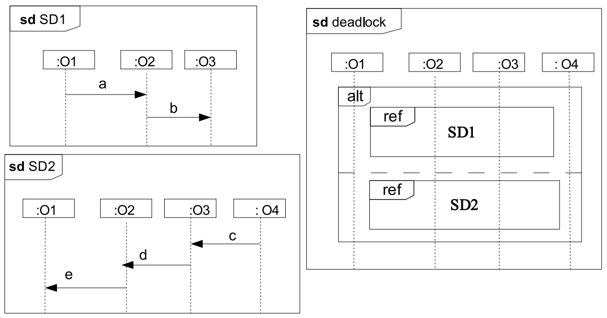

Figure 6. Example leading to deadlock

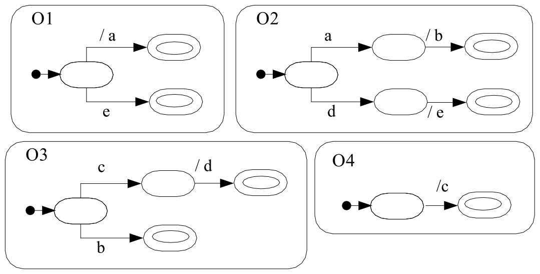

Figure 7. Statecharts generated from Figure 6

4.3. Consequences

First, scenarios cannot be used like a programming language: if the code obtained from initial requirement is not equivalent to what was designed, synthesis is not a way for "executing" scenarios. To solve this problem, [19] proposes to detect scenarios that appear in synthesized model (called "implied scenarios"), and then to enhance the set of requirements to include these implied scenarios. This solution faces two intractable problems: first, detecting if all runs of statecharts are equivalent to runs in the set of requirements is in general an undecidable problem [17]. Hence, the detection of an implied scenario can only be obtained through simulation, and some unspecified runs can be missed. Then, the number of implied scenarios can be infinite, so the set of requirements may never converge towards a stable set of scenarios.

From this consideration, we see synthesis from scenarios as an entry point towards more operational models. If one considers sequence diagrams as sample behaviors of a system, then it is clear that each behavior described must match (modulo a certain abstraction) at least one run of an implementation of this system. If the chosen communication model is adequate, synthesis can ensure this property. The statecharts generated from synthesis must be refined (if possible in a traceable way) to go towards code. Then, scenarios can be used as tests to check that an implementation fulfills the initial requirements.

Note that the difference between the behaviors defined in the requirements and the behaviors of state machines is often due to a loss of synchronization that is implicit in some scenarios. So far, the synthesis approaches have focused more on the behaviors expressed than on the mechanisms needed to implement them. Consider again the example of Figure 6. The implementation of a consensus mechanism between objects $O1$ and $O4$ could ensure that both objects behave exclusively as in scenario SD1 or as in scenario SD2 (up to consensus hiding). Such situations can be detected automatically, and one can imagine that ad hoc solutions (synchronizations, consensus, ...) could also be automatically integrated into synthesized state machines to ensure equality between inter and intra views of the system.

5 Related work

Due to the poor expressive power of UML1.x sequence diagrams, the proposed solutions for statecharts synthesis [10, 11, 16, 21] often use additional information or ad hoc assumptions for managing several scenarios. Whittle et al propose in [21] to augment messages in sequences diagrams with pre and postconditions given in the OCL (Object Constraints Language) which refer to global state variables. State variables identify identical states throughout different scenarios and guide the synthesis process. The drawback of this approach is that causality between events in a SD is not exploited by the synthesis process, except if it is explicitely specified by variables (but such a level of detail is asking too much precision at the requirement stage).

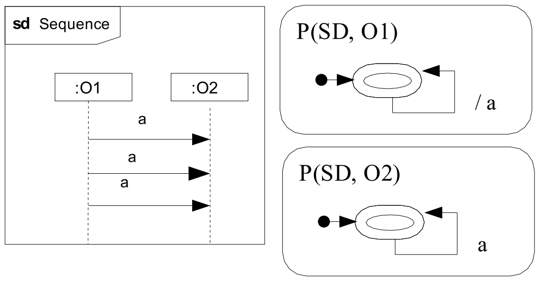

Figure 8. Simple sequence diagram

Consider the example of Figure 8. The same message is sent and received three times. With the approach proposed by *Whittle et al*, the generated statechart would only have one transition corresponding to message emission/reception, except if variables explicitly specify that the system's state evolves. Our approach does not use variables, and structures the state machines and transitions thanks to information provided by lifeline orderings and SD operators. However, the introduction of variables would probably be necessary for state unification in the case of statechart synthesis from several RESD.

Koskimies et al describe a method in [11, 12] to generate flat statecharts from a set of scenarios. It uses the Biermann-Krishnaswamy algorithm [2] which infers programs from traces. This work establishes a correspondence between traces and scenarios and between programs and statecharts. Sending events define states while receiving events define transitions. The main assumption of the approach is that states are identical if they are associated to the same sending event. Again, this state identification may lead to arbitrary merging when the same message can be sent several times. The algorithm proposed by Mäkinen et al [16] is also interactive, and generates flat statecharts from UML sequences diagrams. The main advantage of this approach is to allow interaction with user to accept or to refuse the generated statecharts. The work of Khriss et al. [10] also proposes an interactive algorithm to generate statecharts from multiples scenarios expressed as UML collaborations. To integrate statecharts, the algorithm interacts with users to add state names to the generated statecharts.

Some works study state machines synthesis from Message Sequence Charts (MSC) [9]. MSCs allows composition of basic scenarios (bMSCs) with High-Level Message Sequence Charts (HMSC). This composition mechanism is very close to current SD in UML 2.0. Uchitel [18, 19] proposes to synthesize labeled transition systems from a HMSC. Communications between state machines are synchronous. As shown is Section 4, this can have an important impact on the synthesis process, due to the shape of scenarios that can be used to express requirements, and on the relation between inter object and synthesized intra object views of the system.

Kruger [13] proposes to generate statecharts from a set of MSC. States of the synthesized statecharts are identified using conditions of MSCs. The same condition in several scenarios refers to the same state of a statechart.

The approaches proposed by [1, 15] are based on projection of Message Sequence Charts to obtain SDL code. No restriction is imposed on the initial scenarios, and the SDL behaviors synthesized are not always comparable with the scenarios. A similar approach [14] proposes a synthesis of roomcharts (a kind of asynchronous statecharts) from High-Level Message Sequence charts. This work imposes some strong restrictions on the shape of scenarios used in order to ensure equality between requirements and behaviors of synthesized machines. As discussed in 4, we think that restrictions to scenario often produce poor languages.

6 Conclusion

This paper has proposed an algebraic framework for synthesizing statecharts from UML 2.0 sequence diagrams. Assuming that the statecharts EventDispatcher semantics is that of very generic fifo queues, we established that our synthesis framework ensures the inclusion of initial scenarios in the behaviors of the synthesized state machines. For the moment, our approach is limited to three main operator of UML 2.0

sequence diagrams: **seq**, **alt**, and **loop**. The extension of this framework to include more UML 2.0 operators such as opt (the optional operator) or loops with explicit bounds is currently under study. A prototype of the proposed approach has been implemented in Java. It takes as input interactions specified in textual format (close to [9]), and produces a statecharts for each object. We have used our approach for a complete ATM example including ten basic SDs. The prototype tool is also used on a well known banking system case study [22]. We are currently using this approach in the context of product families.

References

[1] M. Abdalla, F. Khendek, and G. Butler. New results on deriving sdl specifications from mscs. In G. . Y. e. R.Dssouli, editor, *Proc. of 9th SDL forum*, pages 51–66, 1999.

[2] A. Biermann and Krishnaswamy.R. Constrcuting programs from example computations. *IEEE Transaction Software Engineering*, 2(3):141–153, September 1976.

[3] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System design*, 19(1):45–80, 2001.

[4] A. Engels, S. Mauw, and M. Reniers. A hierarchy of communication models for message sequence charts. In T. H. T. Mizuno, N. Shiratori and A. Togashi, editors, *Proc. of FORTE X and PSTV XVII*, pages 75–90, Osaka, Japon, Novembre 1997. Chapman & Hall.

[5] O. M. Group. Uinified modeling language specification version 2.0: Superstructure. Technical Report ptc/03-08-02, OMG, 2003.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[7] D. Harel and R. Marelly. *Come, Let's Play : Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.

[8] L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from hmscs. In *5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, April 2000.

[9] ITU-T. Z.120 : Message sequence charts (MSC), november 1999.

[10] I. Khriss, M. Elkoutbi, and R. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *Proc. of UML'98: Beyond the Notation*, pages 115–126, 1998.

[11] K. Koskimies, T. Systä, J. Tuomi, and Männistö.T. Automated support for modeling oo software. *IEEE Software*, 15:87–94, Janu 1998.

[12] K. M. Koskimies, T. Systä, and J. Tuomi. Sced: A tool for dynamic modeling object systems. Technical Report A-1996-4, University of Tampere, 1996.

[13] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems.* Kluwer Academic Publishers, 1999.

[14] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing room models from message sequence chart specifications. In *Proc. of 13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, Octobre 1998.

[15] N. Mansurov and D. Zhukov. Automatic synthesis of sdl models in use case methodology. In G. . Y. e. R.Dssouli, editor, *Proc. of 9th SDL forum*, pages 225–240, 1999.

[16] E. Mäkinen and T. Systä. Mas-an interactive synthesizer to support behavioral modeling. In *Proc. of International Conference on Software Engineering (ICSE 2001)*, 2001.

[17] A. Muscholl and D. Peled. Message Sequence Graphs and decision problems on mazurkiewicz traces. In *Proc. of MFCS'99*, LNCS 1672, 1999.

[18] S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *Proc. of International Conference on Software Engineering (ICSE 2001)*, 2001.

[19] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *proceedings of the 9th European Software Engineering Conferece and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'01)*, September 2001.

[20] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transaction on Software Engineering*, 29(2):99–115, February 2003.

[21] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. of International Conference on Software Engineering (ICSE 2000)*, 2000.

[22] T. Ziadi. Technical and additional material. http://www.irisa.fr/triskell/results/ICSE04/.