A Workbench for Synthesising Behaviour Models from Scenarios

Sebastian Uchitel and Jeff Kramer
Department of Computing, Imperial College
180 Queen's Gate, London, SW7 2BZ, UK
{su2, jk}@doc.ic.ac.uk

Abstract

Scenario-based specifications such as Message Sequence Charts (MSCs) are becoming increasingly popular as part of a requirements specification. Our objective is to facilitate the development of behaviour models in conjunction with scenarios. In this paper, we first *present an MSC language with semantics in terms of labelled transition systems and parallel composition. The language integrates existing languages based on the use of high-level MSCs (hMSCs) and on identifying component states. This integration allows stakeholders to break up scenario specifications into manageable parts using hMCSs and to explicitly introduce additional information and domain-specific or other assumptions using state labels.* Secondly, *we present an algorithm, implemented in Java, which translates scenarios into a specification in the form of Finite Sequential Processes. This can then be fed to the Labelled Transition System Analyser for model checking and animation.* Finally *we show how many of the assumptions embedded in existing synthesis approaches can be translated into our approach. Thus we provide the basis of a common workbench for supporting MSC specifications, behaviour synthesis and analysis.*

1. Introduction

The software engineering community has long understood the importance of requirements elicitation. Stakeholder involvement in the elicitation process and tools to help build a common ground between stakeholders and developers is essential in order to obtain a good requirements definition. Consequently, it is not surprising that scenarios have become increasingly popular as a requirements specification technique. Scenarios describe how system components (in the broadest sense) and users interact in order to provide system level functionality. Each scenario is a partial story which, when combined with all other scenarios, should conform to provide a complete system description. Thus stakeholders may develop descriptions independently, contributing their own view of the system to those of other stakeholders.

Our objective is to facilitate the development of behaviour models in conjunction with scenarios. Such models are complementary to scenarios. In addition to providing an alternative view, we believe that there is benefit to be gained by experimenting with and replaying analysis results from behaviour models in order to help correct, elaborate and refine scenario-based specifications. We aim to provide a workbench for supporting various approaches to MSC specification, behaviour synthesis and analysis.

Message Sequence Charts (MSCs) [1], together with their UML counterpart Sequence Diagrams [2], are widely accepted notations for scenario-based specification. Nevertheless, up to now, there is little agreement on the exact meaning of these graphical languages. There is, of course, a core semantics on which most approaches coincide, especially in terms of explaining a single scenario. However, when interpreting several scenarios together, we can identify very distinct approaches.

For instance, in the approach adopted by the International Telecommunication Union (ITU) [1] and others [3-5], the focus is on providing MSC specifications with a means for managing complexity. *Basic MSCs* (bMSCs) are used to specify simple sequences of behaviour. *High-level MSCs* (hMSCs) are directed graphs with bMSCs as nodes and edges indicating their possible order. hMSCs allow stakeholders to reuse scenarios within a specification and to introduce sequences, loops, and disjunctions of bMSCs [1]. The advantage of the hMSC approach is that it allows stakeholders to break up a scenario specification into manageable parts in a simple, intuitive, and operational way, and to show how these different parts relate.

Another approach that differs significantly is presented in [6-8]. The focus here is on identifying, throughout the set of bMSCs, those states that are considered to refer to the same component state. As explained by Rudolph et al. [4], the use of hMSCs as the only way of introducing alternative system behaviours forces stakeholders to specify many short bMSCs; whereas, in this approach, complex component behaviour can be shown in bMSCs of any length as information is provided at the component level. The main problem with this approach is that the criteria used for identifying component states are rarely made explicit within the MSC specification. Instead, when constructing a behaviour model for components, the criteria are often embedded into the synthesis algorithms.

For example, Whittle and Schumann [6] use the Object Constraint Language (OCL) to express pre- and post-conditions for messages. These are traversed with bMSCs to produce a valuation of global state variables in bMSC states. These valuations are used to identify equivalent states. Another example is the statechart synthesis algorithm in SCED [8]. This approach employs the domain-specific assumption that the capability of outputting a specific message uniquely identifies the state of a component.

In this paper we present an MSC language that integrates approaches based on hMSCs and on identifying component states. However, instead of assuming specific criteria for identifying component states, we provide a simple mechanism for making this information explicit within an MSC specification using bMSC state labels In this way we aim to provide a workbench for approaches such as [6-8] that allows for explicit additional information (usually in some other formalism such as OCL) and/or domain-specific or other assumptions with an MSC specification. Furthermore, we show how many of these assumptions can be automatically translated into bMSC state labels. MSC semantics is given in terms of Labelled Transition Systems (LTS) and parallel composition [9].

In addition, we provide for the automatic synthesis of system behaviour models from MSCs. We integrate our synthesis process to an existing model checking tool to support system requirements validation. This is done by first translating the MSC specification into a Finite Sequential Processes (FSP) specification [10], which can then be analysed using the labelled transition system Analyser [10] by model checking for deadlock, safety and liveness properties and by model animation [11].

In Section 2 we present the Message Sequence Chart syntax and semantics. The well-known ATM example is used throughout the paper to illustrate the presentation. Section 3 introduces the synthesis algorithm and Section 4 shows how to map some existing approaches to this one. Related work is discussed in Section 5 and the conclusions and future directions of our work are given in Section 6.

2. Message sequence charts

In this section, we briefly describe the syntax and semantics of *Message Sequence Charts* (MSCs). We also introduce the ATM example (see e.g. [8]) which is used to illustrate the different aspects of our approach. This example has several scenarios showing how a customer operates a bank account through an ATM machine and a consortium. For the sake of brevity, we use a reduced set of scenarios.

2.1. Syntax

Syntactically, the language is a subset of the MSC ITU language [1]. A *basic MSC* (bMSC) describes a finite interaction between a set of components (see Figure 1, *Bad Bank Account* for an example). Each vertical line represents a component and is called an *instance*. Each horizontal arrow represents a synchronous *message*, its source on one instance corresponds to a *message output* and its target on a different instance corresponds to a *message input*. Ovals on instances represent *states*, where the label appearing within the oval identifies a particular component state. A state labelled with “Init” is considered to represent the initial state of a component. Existing approaches (e.g. [1]) refer to state labels as conditions, which we consider confusing. Placing MSC events (message inputs, message outputs or state labels) further down on an instance means that they occur later on. For simplicity, throughout the paper, we shall assume that a message label is used for output events by only one component and is used for input events by only one (different) component.

Definition 1. (Instances) An instance is a structure $I = (E, L, <, lbl)$ where

- E is a set of MSC events that is partitioned into sets In, Out, $Cond$ of message input events, message output events and states.
- L is a finite set labels.
- $< \subseteq (E \times E)$ is a total ordering of events. We denote the minimal event e' such that $e < e'$ as $suc(e)$.
- $lbl: E \rightarrow L$ is function that describes each event’s label.

Definition 2. (bMSCs) A bMSC is a structure $B = (I, tgt)$ where

- I is a finite set of bMSC instances $(E_j, L_j, <_j, lbl_j)$ with disjoint events.
- $tgt: \cup In_j \rightarrow \cup Out_j$ is a bijective function that relates output and input events such that
  - If $i \in In_j$ then $tgt(i) \notin Out_j$
  - If $tgt(i) = o$, $i \in In_j$ and $o \in Out_k$ then $lbl_j(i) = lbl_k(o)$.
  - If the transitive closure of $\cup <_i \cup tgt \cup tgt^{-1}$ contains $\{(a,b), (b,a)\}$ then $tgt(a) = b$ or $tgt(b) = a$.

A *high-level MSC* (hMSC) provides the means for composing bMSCs: it is a digraph where nodes are bMSCs and edges indicate their possible continuations (see Figure 1, bottom left, for an example). An hMSC can also have special initial and final nodes that correspond to the initial and final system states. We define hMSCs within the definition of MSC specifications.

Definition 3. (MSC Specification) An MSC Specification is a structure $S = (B, H, C, name)$ where

- B is a finite set of bMSCs (I_j, tgt_j) with disjoint sets of events.
- $H : = B \cup \{Init\} \rightarrow 2^{B \cup \{End\}}$ is the hMSC function that determines the possible continuations of the bMSCs.
- C is a finite set of components.

- *name* is a family of bijective functions $name_j: I_j \rightarrow C$ that determines to which component each instance belongs.

A portion of the MSC specification of the ATM example is shown in Figure 1. It consists of five bMSCs and one hMSC.

The ITU MSC language [1] has several more features. For simplicity we have excluded them, as many are simply syntactical sugar and others do not substantially change any of the results and algorithms that follow. These features are asynchronous messages, queues, co-regions, horizontal composition, inline expressions, actions and global and non-global conditions. There are other features that we have left out as we consider them to be out of scope in this initial stage of our work; however they may be included in the future. These features are timers, gates, process creation and termination, and incomplete messages.

2.2. Semantics

We define the semantics of MSC specifications in terms of labelled transition systems (LTSs) and parallel composition [9]. We first define the semantics of an instance, then go on to that of components, and finally we define the system that is determined by an MSC specification.

There are the two types of information that an MSC specification provides: sequences of message input and outputs, and information on states. Information on sequences of messages is provided by instances. For example, reading from the top to bottom, it is intuitive to say that in the *Bad Bank Account* bMSC of Figure 1 the Consortium inputs a *Verify Account* message, then after sending *Verify Card with Bank*, and receiving a *Bad Bank Account* message, it forwards the message *Bad Account* to the ATM. If each of these events is considered instantaneous, the instance can be viewed as a labelled transition system (LTS) as shown in Figure 2. In addition,

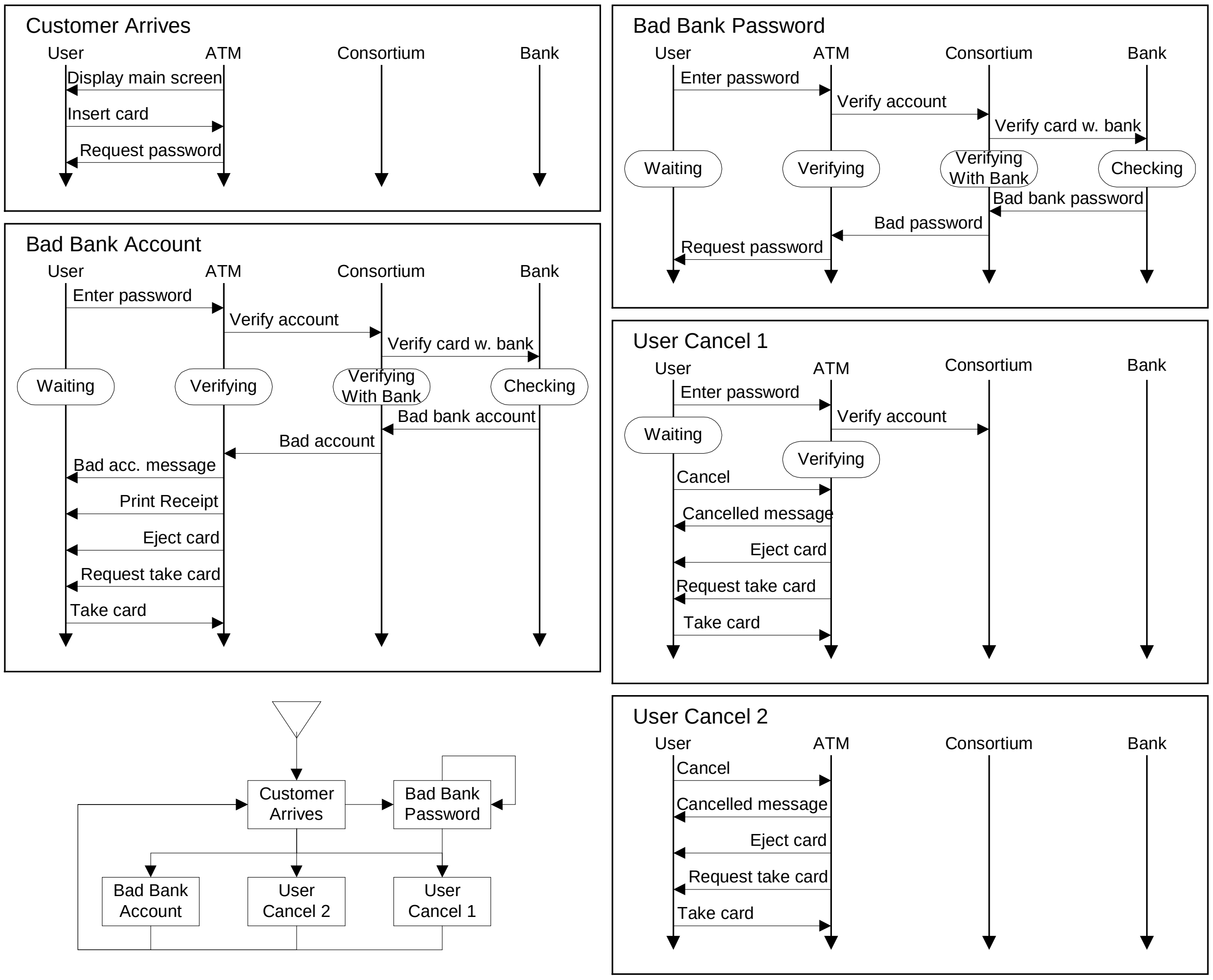

Figure 1 - MSC specification of the ATM system.

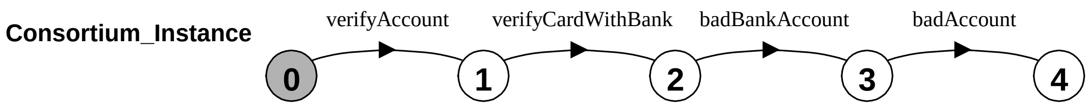

Figure 2 - Instance LTS of Consortium.

state 2 of Figure 2 corresponds to the *Verifying With Bank* state label of the ATM.

To simplify the presentation of semantics, we shall assume normalised instances. A normalised instance is an instance in which there are no consecutive states and no consecutive message events, but in which they alternate, i.e. for all events e, e' such that $e'=suc(e)$ then ($e \in Cond$ and $e' \in \{In \cup Out\}$) or ($e \in \{In \cup Out\}$ and $e' \in Cond$). In addition, it has states as the first and last events, i.e. if e is the minimal or maximal event in E then $e \in Cond$. Normalised instances have events of a special kind, *τ-events*, that represent internal changes of a component's state. Normalising an instance is done by using *τ-events* to separate consecutive states and states labelled with ε to separate consecutive message events.

Definition 4. (Instance LTS of a Component) Let $I = (E, L, <, lbl)$ be a normalised instance. The instance LTS of I is a labelled transition system $(S, A, \Delta, Start)$ where

- $S = Cond$.
- $A = lbl(In) \cup lbl(Out)$.
- *Start* is the minimal event in E
- $\Delta \subseteq (S \times A \times S)$ is the transition relation where $(q \times a \times q') \in \Delta$ if and only if there is a message event $e \in E$ such that $suc(q) = e$, $suc(e) = q'$, and $lbl(e) = a$

We shall refer to the maximal state in E as *Stop*.

State labels and hMSCs provide information on states of instance LTSs. State labels identify component states indicating that, although they appear as distinct in instances, they are actually the same internal component state. For example, there are three different bMSCs in Figure 1 where the ATM reaches a state called *Verifying*. In terms of component behaviour, this means that the ATM could "switch" between bMSCs when arriving at that state. hMSCs provide information on how components can continue once they have completed a bMSC. In other words, they determine a relation between the Start and Stop states of Instance LTSs. For example, according to the hMSC in Figure 1, the states in which components are in once the *Customer Arrives* bMSC concludes, can continue as the initial states of the bMSCs *Bad Bank Password*, *Bad Bank Account*, *User Cancel 1* and *User Cancel 2*. Thus, given an MSC specification, using state labelling and hMSCs, it is possible to define a continuation relation between the instance states of a component plus component initial and final states as follows:

Definition 5. (Continuation Relation) Let $(B, H, C, name)$ be an MSC specification, and $\{I_j = (S_j, A_j, \Delta_j, Start_j)\}$ the set of all instance LTSs of a component c, where I_j corresponds to the instance in bMSC B_j. If q and q' are states in S_j and S_k, the continuation relation of c is $R \subseteq S \times S$ where

- $S = \cup S_j \cup \{Init, End\}$.
- $(q, q) \in R$.
- If $lbl(q) = lbl(q')$ and $lbl(q) \neq \varepsilon$ then $(q, q') \in R$ and $(q', q) \in R$.
- If $B_k \in H(B_j)$, then $(Stop_j, Start_k) \in R$.
- If $B_k \in H(Init)$, then $(Init, Start_k) \in R$.
- If $End \in H(B_j)$, then $(Stop_j, End) \in R$.
- If $lbl(q) = Init$, then $(Init, q) \in R$..

In conclusion, components defined by an MSC specification are the result of putting together their instance LTSs and their continuation relation.

Definition 6. (Component LTS) Let $(B, H, C, name)$ be an MSC specification, $\{I_1, \ldots, I_n\}$ the set of instance LTSs of a component $c \in C$, R^+ is the transitive closure of the continuation relation of c, where $I_j = (S_j, A_j, \Delta_j, Start_j)$. The component LTS of c is a labelled transition system $(S, A, \Delta, Init)$ where

- $S = \cup S_j \cup \{Init, End\}$.
- $A = \cup A_j$
- $(s, a, s') \in \Delta$ if and only if $(q, a, q') \in \Delta_j$ for some j, $(s, q) \in R^+$, $(q', s') \in R^+$.

Finally, the semantics of an MSC specification is the parallel composition of the components it defines.

Definition 7. (System LTS) Let $(B, H, C, name)$ be an MSC specification, $C = \{c_1, \ldots, c_n\}$, and C_i the component LTS of c_i. The system LTS defined by the MSC specification is the parallel composition of all component LTSs: $(C_1 \,||\, \ldots \,||\, C_n)$.

Note that introducing bounded asynchronous communication does not require major change to semantics: port abstractions can be introduced into the final parallel composition as explained in [10].

3. Synthesis of behaviour models

In this section we show how an LTS model can be synthesised from an MSC specification. We translate the MSC specification into a model specification in the form of Finite Sequential Processes (FSP) [10]. We use the Labelled Transition System Analyser [10] for model checking deadlock, safety and liveness properties and for model animation [11].

The synthesis algorithm is outlined in Figure 3. We provide our explanation while applying it to the synthesis of the ATM component of Figure 1. The algorithm first adds state labels to bMSCs to distinguish those states that correspond to starting or stopping of bMSCs: the modified *Bad Bank Account* bMSC is shown in Figure 5.

Second, the algorithm constructs a subset of the relation described in Section 2. Actually, the inverse relation "Is a Continuation of" is built, and only for states, as opposed to all bMSC states. Figure 4 shows each state and the set of states that are a continuation of it.

Third, every instance is broken-up in order to start and finish with a state and to have no states in between. Figure 6 shows the two instances that result from breaking up the ATM instance of the *Bad Bank Account* bMSC.

Fourth comes the actual synthesis. Every instance is trivially translated into an FSP-production by using its first state as the left side of the production and the sequence of events and last state as the right side of the production. Additional productions are constructed by replacing each state with one of the states that can continue it. For the instances in Figure 6, two productions are constructed by the translation, and two from the fact that *E_CustomerArrives* and *E_BadBankPassword* are continuations of *B_BadBankAccount*.

In the fifth step the algorithm generates an FSP specification by merging productions with the same left side into one production using the choice operator. Finally, unreachable non-terminals and duplicate productions are eliminated. The final specification FSP specification, and actual output of our implementation, for the ATM component process is shown in Figure 9. The complete system is the parallel composition of all FSP components:

```
||System = (ATM||Consortium||Bank||User)
```

Once an FSP specification has been generated, LTSA can be used to build an LTS model of each component and of the complete system. Furthermore, LTSA can build minimised models of the FSP specification with respect to observational equivalence [9], which provides a more

```
Input: MSC_Specification S;
Output: FSP_Specification: F;

Relation(State x Set(State) R;
Set(Instance) I;
Set(Production) P;
Component C
Instance i
Production p;
State c, d;

F = emptySet;
1) Add_States_To_Begin_And_End_Of_bMSCs(S);
2) R = Get_IsAContinuationOf_Relation(S);
forAll Component C in S:
   I, P, Q = emptySet;
   3)forAll Instance i of C in S:
      Break up i into instances of the form:
      State, Event*, State;
      Add new instances to I;
   4) forAll Instance i of I:
      Add to P a production (C1: e1->….-> C2)
      forAll state c, d such that (C1, c)
      and (C2, d) are in R:
         Add to P a production (c: e1->….->d)
   5) Merge_Productions_With_Same_Left_Side(P);
   6) Clean_Up(P);
   F = F + OutputInFSPFormat(C, P);
```

Figure 3 - FSP synthesis algorithm.

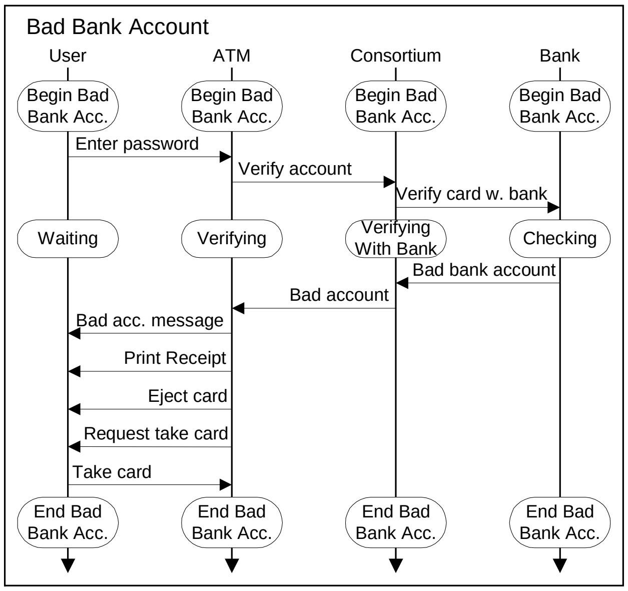

Figure 5 – Modified bMSC for synthesis of ATM.

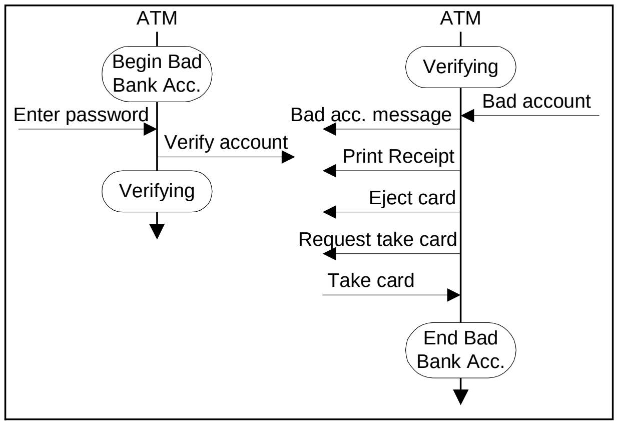

Figure 6 - Broken-up ATM instance

```
E_UserCancel2 = {E_UserCancel2}
E_UserCancel1 = {E_UserCancel1}
B_BadBankAccount = {B_BadBankAccount,
         E_BadBankPassword, E_CustomerArrives}
E_BadBankAccount = {E_BadBankAccount}
B_UserCancel2 = {B_UserCancel2,
         E_BadBankPassword, E_CustomerArrives}
B_UserCancel1 = {B_UserCancel1,
         E_BadBankPassword, E_CustomerArrives}
E_BadBankPassword = {E_BadBankPassword}
Final = {}
B_BadBankPassword = {E_BadBankPassword,
         B_BadBankPassword, E_CustomerArrives}
E_CustomerArrives = {E_CustomerArrives,}
B_CustomerArrives = {E_UserCancel2,
         E_UserCancel1, E_BadBankAccount, Init,
         B_CustomerArrives}
```

Figure 4 – States and their continuations.

```
B_BadBankAccount = (enterPassword ->
                  verifyAccount -> C_Verifying)
E_BadBankPassword = (enterPassword ->
                  verifyAccount -> C_Verifying)
E_CustomerArrives = (enterPassword ->
                  verifyAccount -> C_Verifying)
C_Verifying = (badAccount -> badAccountMessage -
               > printReceipt -> ejectCard ->
               RequestTakeCard -> takeCard ->
               E_BadBankAccount).
```

Figure 7 - Productions for instances of Figure 6.

```
minimal ATM = Init,
E_CustomerArrives = (cancel -> canceledMessage
      -> ejectCard -> requestTakeCard ->
      takeCard -> Init | enterPassword ->
      verifyAccount -> C_Verifying),
Init = (displayMainScreen -> insertCard ->
      requestPassword -> E_CustomerArrives),
C_Verifying = (cancel -> canceledMessage ->
      ejectCard -> requestTakeCard -> takeCard
      -> Init | badPassword -> requestPassword
      -> E_CustomerArrives | badAccount ->
      badAccountMessage -> printReceipt ->
      ejectCard -> requestTakeCard -> takeCard
      -> Init).
```

Figure 9 - FSP specification for ATM component.

compact model and potentially clearer insight into its behaviour. In Figure 8 we show a minimised LTS of the ATM generated by LTSA. It is worth noting the impact that the state *Verifying* has had on the resulting model of the ATM component. Stakeholders have explicitly stated that the state in which the ATM is in after sending a *Verify Account* Message is the same whatever bMSC is being executed. This fact can be seen in Figure 8 where state 5 represents the *Verifying* state.

Since the synthesis algorithm preserves the semantics of the MSC specification [12], the synthesised model can be analysed to provide sound and useful feedback to those who wrote the MSC specification. An immediate result when the complete ATM example is analysed is that the system may deadlock. In Figure 10 LTSA shows a trace that takes the system to deadlock: if the User cancels just after entering a password but before receiving an answer, the ATM, which has requested the account to be verified, does not wait for the answer from the Consortium. Eventually when the ATM serves the User again, it cannot

```
Progress violation for actions: . . .
Trace to terminal set of states:
       displayMainScreen
       insertCard
       requestPassword
       enterPassword
       verifyAccount
       cancel
       canceledMessage
       ejectCard
       requestTakeCard
       takeCard
       displayMainScreen
       insertCard
       requestPassword
       enterPassword
       verifyCardWithBank
       badBankPassword
Actions in terminal set:
       {}
```

Figure 10 - LTSA output with deadlocking trace.

communicate with the Consortium as the latter is still trying to communicate the results of verifying the previous account.

3.1. Implementation

The synthesis algorithm is implemented in Java. It inputs MSC specifications in textual format [1] and outputs an FSP specification. We intend to embed the implementation into LTSA. The implementation, together with the examples used throughout this paper, is available at [13].

As our synthesis algorithm builds component specifications one at a time, computational complexity is not a critical issue; however, the number of states and the size of the hMSC specification could have an important impact on the number of synthesised productions and clean-up procedures. We show that this impact is low in Table 1 where the execution times for the three examples presented in this paper are shown. All examples were run on a Pentium III, 300Mhz, 256Mb with Windows NT 4.0 and Java 1.3.

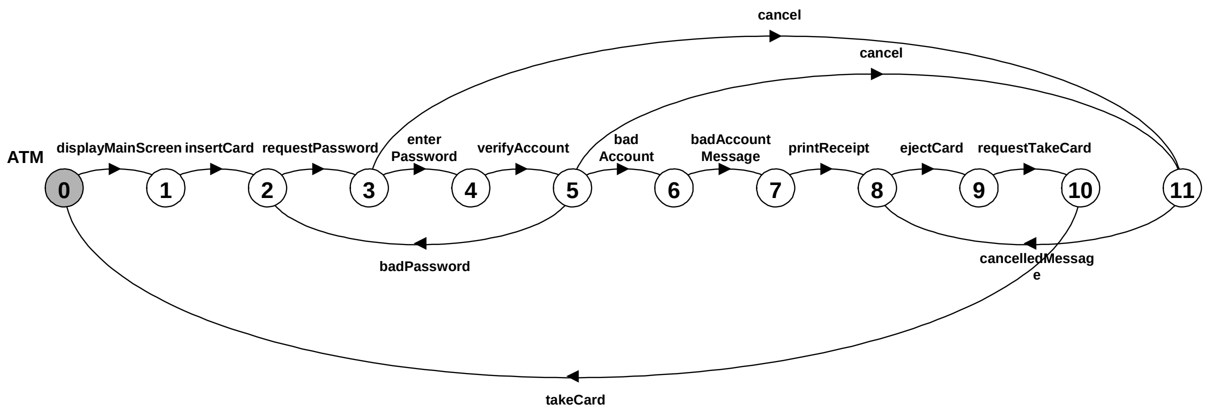

Figure 8 - Labelled transition system for ATM.

Table 1 - Synthesis algorithm execution times.

	Size of hMSC	State Labels	States	Time (ms)
ATM Section 2	11	4	10	181
ATM Section 3.1	15	21	125	200
ATM Section 3.2	4	33	98	171

4. A workbench for synthesis of behaviour models

So far we have presented an MSC specification language that integrates approaches based on hMSCs and on identifying component states. We have also provided a synthesis algorithm that generates LTS behaviour models. In this section we illustrate how this approach can be used as a workbench for experimenting with different approaches for behaviour model synthesis.

Many synthesis procedures have been proposed and, although they agree on the basic interpretation of MSCs, they differ greatly in terms of the algorithms and the results obtained. This is because these approaches embed assumptions in their synthesis algorithms. These can be domain-specific assumptions, assumptions on how to include additional information provided in alternative specification languages or other assumptions based on, for example, characteristics of the stakeholders, organization or development process. Synthesising behaviour models with certain assumptions in mind is not a problem; however, having assumptions embedded in the algorithms results in less flexibility. In this section we show how our approach can be used as a common workbench for these approaches by making assumptions explicit using states. We demonstrate this for two approaches: firstly where additional information is used to determine equivalent states between scenarios [6, 7]; the secondly [8] where it is assumed that the capability of outputting a particular message uniquely identifies the state of the component.

4.1. Additional information and process-specific assumptions

Whittle and Schumann [6] present an algorithm for automatically generating UML statecharts from scenarios combined with a set of message pre- and post-conditions given in UML Object Constraint Language (OCL). In fact, a labelled transition system is first synthesised and then some abstraction techniques are used to build statecharts. LTSs are synthesised by combining information of MSCs and OCL: a valuation for every state of the bMSCs is inferred from the OCL specification. These valuations are used in two different ways:

- Within an instance, two states with the same valuation determine a loop. As not all message occurrences provoke a change in valuation it is also required that valuations be the result of a state-changing message.
- Between instances, two states with the same valuation and a common incoming message label are considered to be referring to the same state.

In addition, there is an underlying assumption: stakeholders describe system behaviour from its initial state.

In order to place this approach in our setting and make

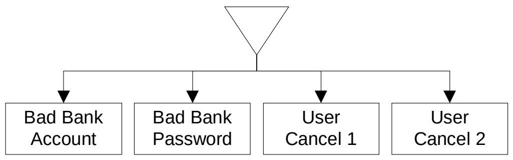

Figure 11 - Explicit specification of initial system state

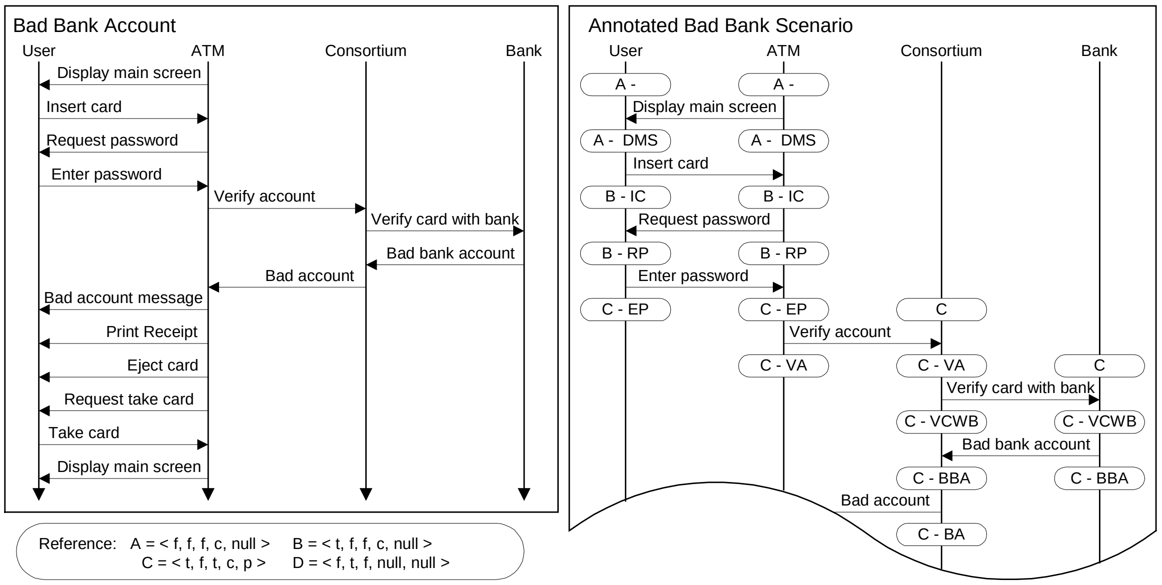

Figure 12 - Original and annotated bMSCs

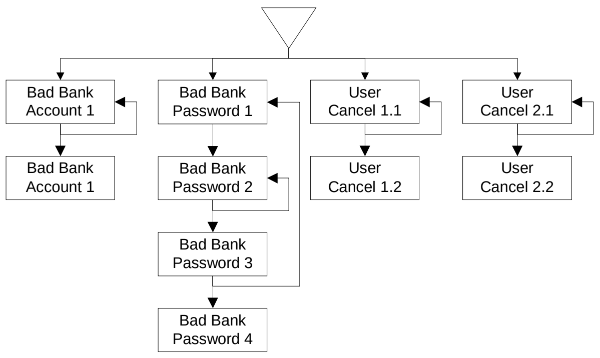

Figure 13 – Explicit specification of detected loops.

all assumptions explicit, we address each of the three issues mentioned above separately. First of all an hMSC is constructed to make explicit the assumption that scenarios describe system behaviour from the very start of it (see Figure 11).

Second, after valuations have been inferred from the OCL specification, and loops have been detected, all states in bMSCs are labelled with the message that precedes it and its inferred valuation. In Figure 12 we show bMSC *Bad Bank Account* as provided in [6] and part of the same bMSC annotated as explained above. Valuations are shown as vectors where each position represents the value of a variable in the OCL specification.

Finally, every loop, detected using the algorithm provided in [6], is used to split bMSCs into three: the initial part that occurs before the loop, the looping part, and the part that occurs after the loop is exited. The hMSC is modified to reflect the relation between the new bMSCs resulting in an hMSC as in Figure 13.

The final MSC specification has all the information in it and all assumptions have been made explicit through a process that can be automated. In other words we have shown that both the ideas and algorithms developed in [6] can be used with our approach giving the benefit of a potentially standardised synthesis algorithm and placing all information in one specification. It is worth noting that the resulting LTS corresponding to the ATM component turns out to be the same as the one for the previous specification (Figure 8). The complete example can be found in [13].

4.2. Domain-specific assumptions

Systä [8] presents the statechart synthesis algorithm implemented in SCED. The algorithm is based on the BK-algorithm [14] for constructing programs from example computations. In the synthesised statecharts, states are labelled with component actions (message outputs) and transitions with a sequence of events (message inputs). In addition, all states that represent the same action are merged as long as they do not introduce non-determinism.

The underlying rationale for this approach is the

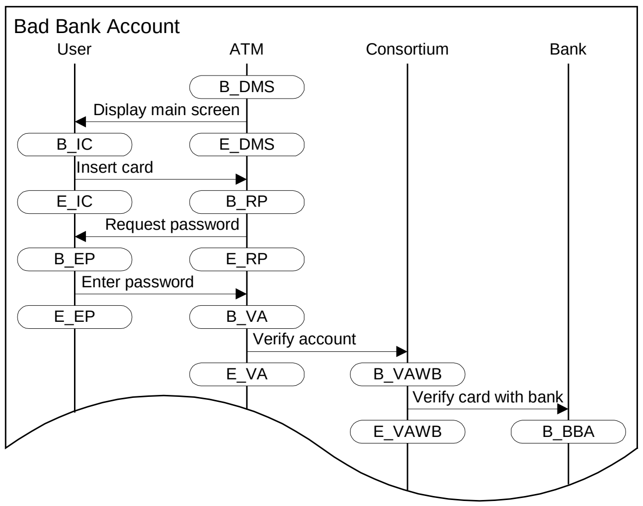

Figure 14 - Annotated bMSC.

assumption that scenarios describe a particular type of component in which the capability of outputting a particular message uniquely identifies its state . This assumption can be made explicit in an MSC specification using state labels. However, there are some minor aspects of the approach that cannot be completely mapped into our setting. Firstly, in [8] resulting statecharts have no initial state. This may not be appropriate, so we prefer to be explicit using an hMSC as shown in Figure 11. Secondly, in [8] non-deterministic models are prohibited as a way of avoiding unwanted component behaviours [15]. We see no need for imposing such a constraint particularly when tools, such as LTSA, for analysing and detecting such behaviours are available. Besides, non-determinism may be desirable and deliberately introduced by designers in some cases.

In conclusion, modulo non-determinacy and initial states, we map the assumptions that are used in the approach by adding two sets of states. To merge states with the same enabled output messages, a state label *B_<Message Label>* is added to component instances just before outputting any message. To model statechart states that represent actions, a state label *E_<Message Label>* is added to component instances just after outputting any message. In Figure 14, the bMSC of Figure 12 is annotated as explained before. Thus, using state labels, an MSC specification can be built that has all the information on how states are supposed (according to [8]) to be merged. The complete example can be found in [13].

5. Related work

Several semantics for scenario-based languages have been proposed, and also a number of synthesis techniques for building models from a scenario description have been developed. Our work focuses on integrating some of these approaches and on providing a workbench for supporting these and future approaches.

There are many approaches that generate statechart models from MSCs [6, 8, 16]. Authors argue that statecharts provide a more structured, and therefore understandable, view of behaviour. Automatically synthesising this structure does require that some design decisions be embedded into the synthesis process. However, we argue that this can be counter-productive. Design decisions should be explicit and changeable, particularly as they may vary a great deal according to the system, designer, and organization. Our approach gives special importance to producing analysable models and uses standard minimisation techniques to help to provide compact, comprehensible models.

Many approaches do not explicitly provide semantics for the scenario language they use, providing instead a synthesis algorithm to some other notation. Broy et al. [16] present a statechart synthesis algorithm in which interpretation of conditions is similar to our use of state labels. However their approach does not support hMSCs. We share the authors' view of MSC specifications as an *exact* representation of interaction sequences, and also the synchronous communication setting.

Whittle and Schumann [6] focus on synthesising readable and understandable statecharts. Besides the LTS synthesis discussed previously, they provide a means of introducing structure and hierarchy into the synthesised model. Although the use of additional information in terms of OCL specification provides interesting feedback of possible specification errors, and the whole approach provides a bridge between initial MSC specifications and more complex specification techniques, the approach tends to be obscured by the assumptions that are embedded into the synthesis algorithms. We have shown how these assumptions can be made explicit, thereby providing a clearer and more transparent basis for their work. Somé et al. [7] also use additional information to infer equivalences between states. We believe that they too might benefit from being mapped onto our approach to make the results of the inference process explicit. Systä's approach [8] discussed in previous sections also focuses on synthesising readable and understandable statecharts. The approach has strong assumptions embedded into the synthesis algorithm as to when states should be merged. We have shown how these assumptions can also be made explicit by mapping the approach to ours.

Some approaches give a different semantics to hMSCs and state labels. The latter are often called conditions [5, 16] [1] and Rudolph et al. [4] allow them in hMSCs for referencing system states as opposed to component states. A referenced bMSC in an hMSC must have the same initial and final global states as its reference. The redundancy introduced by conditions does not provide an alternative view of information; rather it provides a double check for consistency between bMSCs and hMSCs. In addition, as pointed out by the authors, all alternatives must be placed in the hMSC, as choices are not allowed in bMSCs. This leads to short bMSCs. The way we use conditions (state labels) addresses this, as it allows many ways of expressing the systems behaviour, using long or short, and several or few bMSCs as appropriate.

The formal semantics of MSCs proposed by Cobens et al. [5] is part of the Z.120 recommendations for MSCs. The semantics of hMSCs differs slightly as a *late decision* assumption is used. Late decision means that a component, when choosing between two different possible scenarios, will postpone the decision if both scenarios have common initial events. In our approach this needs to be explicitly stated using state labels. The advantage of the late decision assumption is that it can reduce the size of specifications. However we again prefer to make this assumption explicit. Late decision semantics could be translated automatically into state labels in our workbench. Furthermore, the Z.120 formal semantic definition is given in terms of process algebra, with non-standard operators of delayed choice and delayed parallel composition. We prefer the more standard model of LTS with parallel composition. Other similar formalisations to [5] are given in [17-19], however only bMSC semantics is given.

Van Lamsweerde et al. [20] present a different approach to synthesis. A set of examples and counterexamples expressed as scenarios is used to infer a temporal logic specification. Thus, generating explicit declarative requirements from an operational description. Combining these requirements with LTS models may be an interesting possibility for future work.

Alur et al. [21] give the semantics of bMSCs in terms of a partial order of events occurring in the whole system, as opposed to considering one component at a time. The focus is on characterizing and providing algorithms for checking satisfiability, and weak and strong realizability. As in our view, MSCs determine a unique set of system components rather than a set of system traces, issues such as satisfiability and weak realizability do not apply, while strong realizability (essentially deadlock freedom) can be verified using LTSA. However, we are currently looking into MSCs with semantics based traces and analysing the scenarios implied by a model that satisfies the MSC specification. Finally, Ben-Abdallah et al. [22] focus on detecting process divergence and non-local choice, for which efficient algorithms are given.

6. Conclusions

We have presented a language for MSC specifications that integrates approaches based on hMSCs and on identifying component states. We have implemented a synthesis algorithm that generates LTS models for model analysis in LTSA, and illustrated how this approach can be used as a workbench for model synthesis and analysis.

Using hMSCs we help to manage complexity of MSC specifications, promoting scenario reuse, and providing a simple, intuitive, operational way of showing how scenarios relate. Using state labels to provide information on component states we help to make explicit any additional information, and domain-specific or general assumptions in MSC specifications. By generating FSP specifications, our approach integrates with LTSA, thus supporting model checking of deadlock and safety and liveness properties. There is also the potential for model animation [11] as a means of including further domain constraints and of making the models more comprehensible to stakeholders and developers. Finally, by taking two dissimilar approaches with their own assumptions and their own means of adding information to MSC specifications, and by showing how they can be built using our approach, we have indicated how our approach could serve as a workbench for other synthesis approaches.

Scenarios have proved to be a good tool for bridging the gap between stakeholders and developers. However, up to now, this is mainly a one-way bridge in which developers gain more insight of stakeholders' domain knowledge. Future work will be focused on building a bridge in the other direction, i.e. building mechanisms to provide feedback of the developer's world to stakeholders. Preliminary work in this direction is promising. We are automating the construction of alternative system views from synthesised LTS models. Interestingly, many views can be generated by taking advantage of the semantic overlap between hMSCs and state labels. The latter identify component states across scenarios, while the former provide information about all components by relating bMSCs. Moving information from state labels to hMSCs allows for a large number of possible views that vary from long bMSCs that start at the system's initial state to short bMSCs that optimise reuse. These views can allow stakeholders to gain more insight into their own MSC specifications or be used by designers to show the impact of their changes to behavioural models in a language that stakeholders manage.

Acknowledgements

We would like to thank Jeff Magee for his helpful comments and discussion.

References

1. ITU, *ITU-T Recommendation Z.120. Message Sequence Charts (MSC'96)*. 1996, ITU Telecommunication Standardisation Sector, Geneva.
2. Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, ed. Addison-Wesley. 1998.
3. Alur, R., G.J. Holzmann, and D. Peled. *An Analyser for Message Sequence Charts*. *2nd Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, 1996, Passau, Germany.
4. Rudolph, E., P. Graubmann, and J. Grabowski. *Tutorial on Message Sequence Charts '96*. *FORTE/PSTV*, 1996, Kaiserslautern, Germany.
5. Cobens, J.M.H., *et al.*, *Formal Semantics of Message Sequence Charts*. 1998, Eindenhoven University of Technology, Eindhoven, The Netherlands.
6. Whittle, J. and J. Schumann. *Generating Statechart Designs from Scenarios*. *22nd Intl. Conference on Software Engineering*, 2000, Limerick, Ireland.
7. Somé, S., R. Dssouli, and J. Vaucher. *From Scenarios to Timed Automata: Building Specifications from User Requirements*. *Asia Pacific Software Engineering Conference*, 1995.
8. Systä, T. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. *Dept. of Computer and Information Sciences*, University of Tampere, 2000.
9. Milner, R., *Communication and Concurrency*. International Series in Computer Science, 1989, Prentice-Hall.
10. Magee, J. and J. Kramer, *Concurrency: State Models and Java Programs*. 1999, New York: John Wiley & Sons Ltd.
11. Magee, J., *et al. Graphical Animation of Behaviour Models*. *22nd International Conference on Software Engineering*, 2000, Limerick, Ireland.
12. Uchitel, S. and J. Kramer, *A Sound Algorithm for Synthesis of Behaviour Models from Scenarios*. 2001, Department of Computing, Imperial College. London.
13. Uchitel, S., *MSC-FSP Synthesiser*. 2000, Available at http://www-dse.doc.ic.ac.uk/~su2/Synthesis/.
14. Biermann, A.W. and R. Krishnaswamy, *Constructing Programs from Example Computations.* IEEE Transactions on Software Engineering, 1976, **2**(3): p. 141-153.
15. Koskimies, K. and E. Mäkinen, *Automatic Synthesis of State Machines from Trace Diagrams.* Software-Practice and Experience, 1994, **24**(7): p. 643-658.
16. Broy, M., *et al. From MSCs to Statecharts*. *Distributed and Parallel Embedded Systems*. 1999: Kluwer Academic Publishers.
17. Heymer, S. *A Non-Interleaving Semantics for MSC*. *1st Workshop of the SDL Forum Society on SDL and MSC*. 1998, Berlin, Germany.
18. Katoen, J.-P. and L. Lambert. *Pomesets for Message Sequence Charts*. *1st Workshop of the SDL Forum Society on SDL and MSC*. 1998, Berlin, Germany.
19. Mauw, S. *The Formalization of Message Sequence Charts*. *1st Workshop of the SDL Forum Society on SDL and MSC*. 1998, Berlin, Germany.
20. Van Lamsweerde, A. and L. Willemet, *Inferring Declarative Requirements Specifications from Operational Scenarios.* IEEE Transactions on Software Engineering, 1998, **24**(12): p. 1089-1114.
21. Alur, R., K. Etessami, and M. Yannakakis. *Inference of Message Sequence Charts*. *22nd International Conference on Software Engineering*. 2000, Limerick, Ireland.
22. Ben-Abdhallah, H. and S. Leue. *Syntactic Detection of Process Divergence and Non-Local Choice in Message Sequence Charts*. *3rd Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 1997, Springer-Verlag.