Enriching SDL Specifications with MSCs*

Ferhat Khendek[1] and Daniel Vincent[2]

[1]Department of Electrical and Computer Engineering
Concordia University
1455, de Maisonneuve W., Montréal (P.Q.),
Canada H3G 1M8
E-mail: khendek@ece.concordia.ca
URL: http://www.ece.concordia.ca/~khendek

[2]France Telecom R&D, Lannion, France

Abstract

Software systems evolve through addition of new behaviors and new components to the existing behavior and architecture. In this paper, we address the issue of enriching automatically SDL specifications with new behaviors specified with MSCs. The new behaviors should not inhibit (invalidate) existing and already validated behavior. In other words, the existing behavior should be preserved in the new SDL specification. We define formally the notion of behavior preservation in the context of SDL, taking into account its semantics, and we illustrate our SDL specification enrichment approach through an example.

Keywords: SDL, MSC, Extension, Behavior Comparison, Incremental Design, *MSC2SDL*.

1. Introduction

Formal techniques can play an important role in all the phases of a software process. Among these techniques, MSC (Message Sequence Charts) [1] and SDL (Specification and Description Language) [2] are very popular and widely used. MSC and SDL are supported by powerful commercial tools, such as ObjectGeode [3] and SDT [4], for simulation, verification, code generation and test cases generation. So far, they have been used successfully in the requirement, design, implementation and testing phases.

During the maintenance phase, software systems evolve, bugs are discovered and corrected, algorithms used in the code are improved, and new features (behaviors) are added. In this paper, we focus on the third kind of maintenance, which is referred to software extension or enrichment. Instead of enriching the end product (code), we propose to enrich the SDL specification, which represents the reference (the basis) for code and test cases generation. More specifically, we propose to look at the following problem: "Given an SDL specification and a new behavior specified with an MSC, how to enrich the SDL specification with this new behavior and preserve the existing behavior ?"

* Partially supported by France Telecom.

Incremental approaches have been introduced in the context of SDL/MSC methodology [12] and LOTOS [13, 14], which is a synchronous language for distributed system specification. In this paper, we first discuss a theoretical question, which is the meaning of SDL behavior preservation, before we introduce our approach for enriching SDL specifications with behaviors specified with MSCs.

The rest of the paper is structured as follows. Section 2 discusses the notion of behavior preservation in the context of SDL semantics, and introduces a new relation for comparing SDL behaviors. Section 3 describes our approach for enriching SDL specifications with new behaviors through an example. In Section 4, we conclude.

2. Behavior Preservation in SDL

In order to compare and relate SDL specifications to each other, we need to define the notion of behavior preservation. Informally, we say that an SDL specification S2 *preserves*, or *extends*, the behavior of another SDL specification S1, if and only if S2 exhibits all the behaviors of S1 without any new non-determinism. In other words, S2 shows all the traces (sequences of interactions with the environment) of S1 without any new "surprise" to the environment relatively to these traces. S2 can be used wherever S1 can be used. S2 may exhibit more behaviors than S1. For instance, S2 may take into account new input signals and define the behavior for these signals. S2 may also react with more output signals for already defined input signals.

Several conformance and equivalence relations have been defined in the formal techniques literature, especially for transition systems [5, 6]. These relations are based on blocking semantics. In other words, they are based on semantics where a system or a process blocks for a non-defined interaction with its environment. These relations cannot be applied for SDL, which has a non-blocking semantics. An SDL process communicates with its environment in an asynchronous manner. The environment sends signals through channels and these signals are queued in the input queue of the process. At a given state, an unexpected input signal, i.e. a signal not explicitly specified as input, is just removed from the process queue. Furthermore, SDL processes may *save* signals for future consumption.

In order to define behavior preservation, or the extension relation, in the context of SDL, we first introduce few notations without referring to any underlying semantic model for SDL. However, our definitions can be understood and applied under any extended input/output communicating finite state machine model for SDL.

- For a process *P*, a state *s*, *Input(s)* is defined as the set of input signals for which an explicit transition is defined from *s*. In this definition we view the SDL keyword *none* as an input signal.

- For a process *P*, a state *s*, *Save(s)* is defined as the set of input signals saved in state *s*.

- For a process *P*, a state *s*, an input *i*, *Output(s, i)* is defined as the set of sequences of output signals in the transitions initiated by input signal *i* from state *s*. The "." is used for concatenating sequences.

- For a process *P*, a state *s*, an input *i*, and a sequence *seq* in *Output(s, i)*, *NextState(s, i, seq)* represents the set of states that can be reached from *s* with input *i* and output sequence *seq*. Notice that in SDL, with a given input signal and a sequence of output signals we can reach different states. This kind of non-determinism can be modeled easily with the SDL keyword *any*.

- A sequence *seq2* of (output) signals *extends* another sequence *seq1* of (output) signals, if and only if *seq1* is a prefix of *seq2*. For instance, sequence *0.1.1.0 extends* sequence *0.1*.

In this paper, we only consider the input/output signals for the definition of the extension relation. In other words, our definition does not take into account actions, procedures, etc. that can be defined in the transitions. In the following, we first define the extension relation between a pair of states. Informally, we say that state s_2 extends state s_1, when s_2 has all the explicit inputs of s_1 (and more) and for each of them, for each state s_1' we can reach from s_1 and an output sequence *seq1*, there is a state s_2' we can reach from s_2 and an output sequence *seq2* with s_2' *extends* s_1' and *seq2 extends seq1*, and for each state s_2' we can reach from s_2 and an output sequence *seq2*, there is a state s_1' we can reach from s_1 and an output sequence *seq1* with s_2' *extends* s_1' and *seq2 extends seq1*, and a signal saved in s_1 is also saved in s_2. The extension between processes is defined as the extension between their initial states.

Definition 1 (State Extension)
State s_2 *extends* state s_1, if and only if

- $Input(s_1) \subseteq Input(s_2)$,
- $Save(s_1) \subseteq Save(s_2)$,
- $\forall i \in Input(s_1)$, $\forall seq1 \in Output(s_1, i)$, $\exists seq2 \in Output(s_2, i)$, such that
  - o *seq2 extends seq1,* and
  - o $\forall s_2' \in NextState(s_2, i, seq2)$, $\exists s_1' \in NextState(s_1, i, seq1)$ such that s_2' *extends* s_1', *and*
- $\forall i \in Input(s_1)$, $\forall seq2 \in Output(s_2, i)$, $\exists seq1 \in Output(s_1, i)$, such that
  - o *seq2 extends seq1,* and
  - o $\forall s_2' \in NextState(s_2, i, seq2)$, $\exists s_1' \in NextState(s_1, i, seq1)$ such that s_2' *extends* s_1'. ■

Definition 2 (Process Extension)
A process *P2 extends* a process *P1*, if and only if
Initial(P2) extends Initial(P1), where
Initial(Pi) represents the initial state of *Pi*, with i = 1, 2. ■

To illustrate the extension relation between processes, consider the example given in Figure 1. Both processes, *P1* and *P2,* are non-deterministic. As we can see, state *s4* in *P2* extends state *s1* in *P1*, state *s5* in *P2* extends *s2*. Now , if we compare states *so* and *s3*, we see that *Input(so) = {a}* and *Input(s3) = {a}*, *Output(so, a) = Output(s3, a) = {0.1, 1}*, *NextState(so, a, 0.1) = {s1}*, *NextState(s3, a, 0.1) = {s4}*, *0.1* extends itself and *s4* extends *s1*, *NextState(so, a, 1) = {s2}*, *NextState(s3, a, 1) = {s5}*, *1* extends itself and *s5* extends *s2*. In state *so*, process *P1* does not save any signal, while process *P2* saves signal *b*. We can conclude that *s3* extends *so*. Therefore, *P2* extends *P1*.

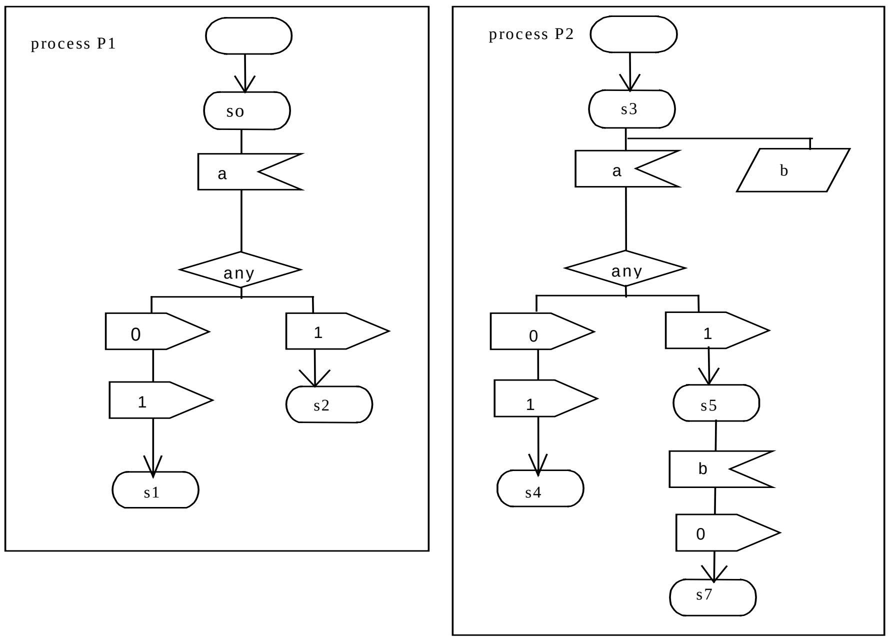

Figure 1. P2 is an extension of P1.

Now, if we consider the second example given in Figure 2, we can see that process *P4* is not an extension of *P3*. The reason is that *P4* introduces a new non-determinism in state *s3*. Indeed, after consuming signal *a*, the output can be non-deterministically *0.1* or *1* for *P4*. Process *P3* always produces *0.1* after consuming *a*. Of course process *P4* has more behaviors than process *P3*, but because of this new non-determinism, *P4* is not an extension of *P3*.

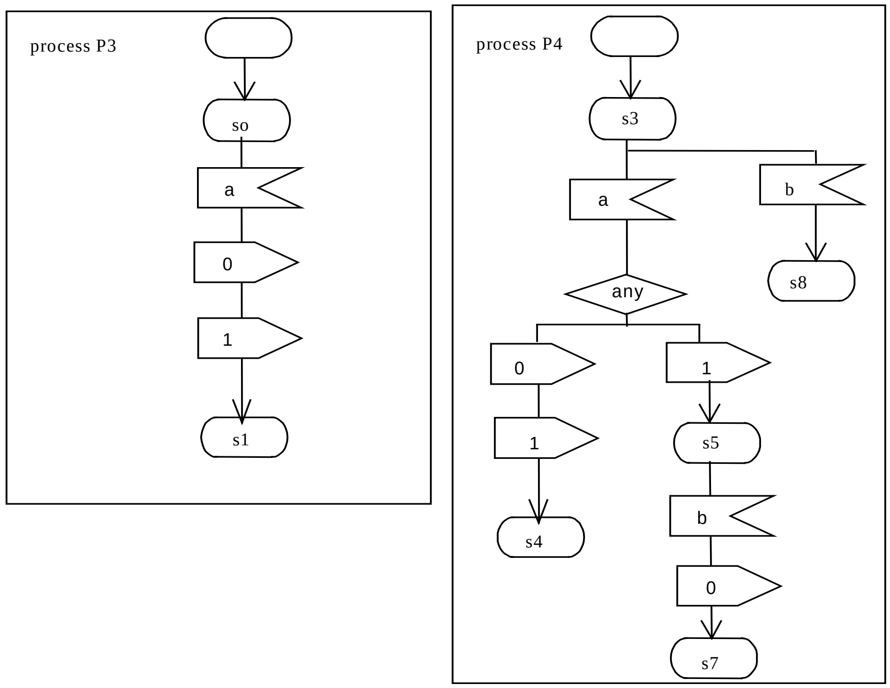

Figure 2. P4 is not an extension of P3.

We are interested in extending SDL system specifications. Let us consider a SDL specification S and denote by *Input(S)* the set of signals that can be sent from the environment to S and by *Output(S)* the set of signals that S can send to the environment. From the environment point of view, i.e. hiding all the internal interactions between the processes in the system, the behavior of a SDL system specification S can be seen, similarly to the processes, as an input/output finite state machine. A state of the system is defined by the states of the processes in the system. The system interacts with its environment through signals from *Input(S)* and *Output(S)*. The transitions are triggered by signals from Input(S) or by *none*, during a transition a sequence of output signals from *Output(S)* is sent to the environment and the system S moves from one state to a next state in a very similar manner to a process. The set of signals to be saved in a global state *s*, *Save(s)*, is defined as the union of the saves of the states of the processes that define this global state. With this abstract view of the system in mind, we define the extension relation between global states in a very similar way to Definition 1 and the extension relation between SDL systems in a very similar manner to Definition 2.

In order to extend a SDL system specification, our approach consists of extending the processes in the system. We claim that extending the processes in the system is a sufficient condition to extend the whole system, provided that the sets of input signals for the processes in the system are disjoint. However, this is not a necessary condition. Further theoretical study of the congruence property of the extension relation is left for future investigations.

3. Enriching SDL Specifications

It is very common that new behaviors or new components have to be added to an existing SDL specification, or SDL specification are to be designed in a step-wise manner [9, 10, 11, 12]. Enriching an SDL specification with a new MSC may lead into different situations depending on the MSC specification. Among the different possible scenarios, we can list the following:

- New processes are specified in the given MSC (i.e. process types not already defined in the SDL specification).

- New communications between existing instances or between new and old instances.

- New behaviors to take into consideration for some processes.

The first two cases will require the enrichment of the system architecture. Indeed, if we have at least a new process, we have to specify clearly where this process goes in the architecture and how it communicates with the other processes. New interactions between two existing (or between existing and new) instances may require the addition of new channels or enrichment of existing channels with new signals if possible.

In this paper, we illustrate our approach for enriching SDL specifications with MSCs using the ATM (Automatic Teller Machine of a Bank) example.

Initial ATM specification

The initial ATM specification consists of a block named *ATM_Block*, which consists of the *ATM* process and the *Bank* process. Figure 3 shows the *ATM_Block* and its processes, channels and signals. The ATM system interacts with users in the environment. In this specification, we initially have the *Withdraw* function and the possibility of ending a transaction. In this specification, the user supplies a card and a password to the ATM machine. Then, he takes one of two options: withdraw allowable amount from his account or end the transaction. However, the user card may be rejected for the following reasons: password has not been supplied or incorrect password.

The behaviors of the processes *ATM* and *Bank* are given in Figure 4 and Figure 5, respectively. The process Bank is internal to the system. It does not interact with the environment. The process *ATM* interacts with the users in the environment. Our specification assumes one user at a time, since we only have one *ATM* process.

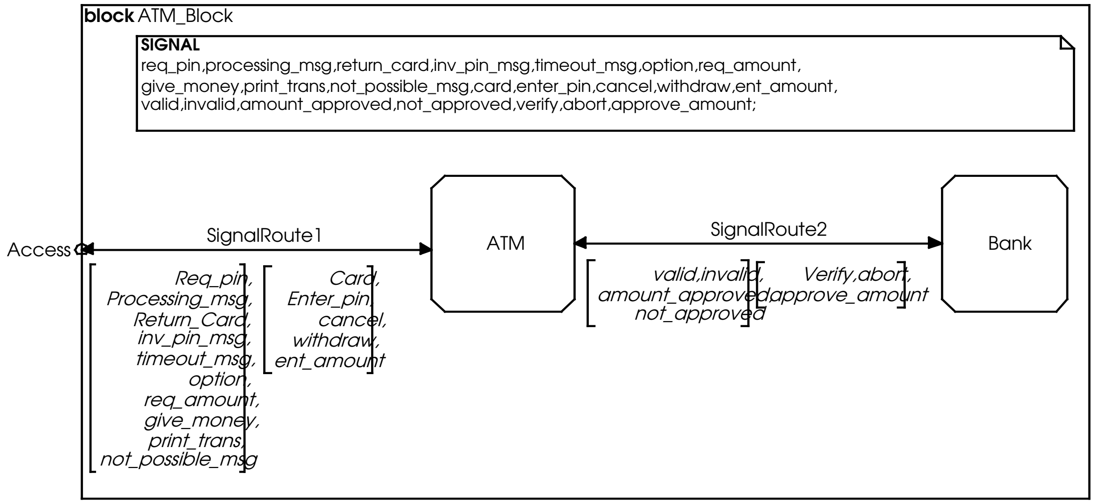

Figure 3. ATM architecture specification (*ATM_Block*).

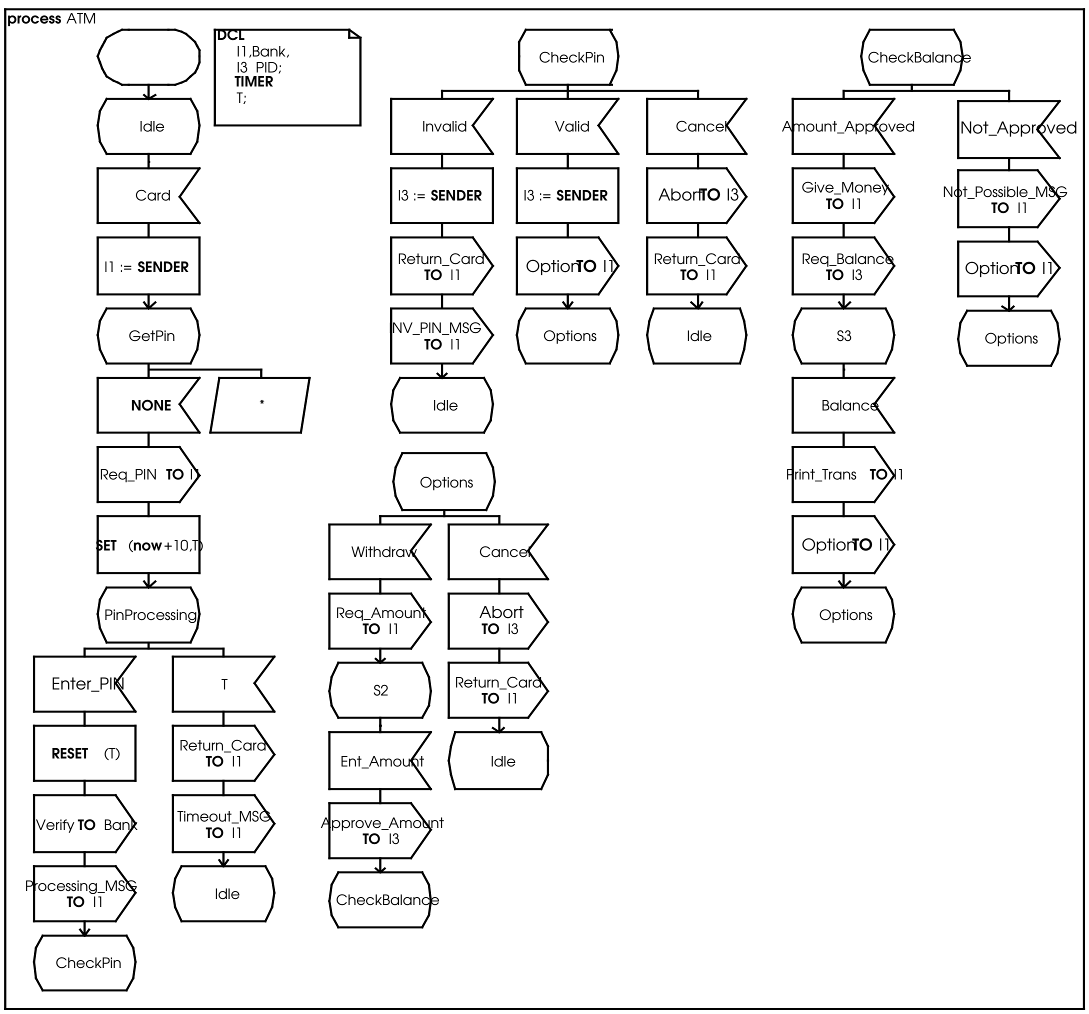

Figure 4. *ATM* process.

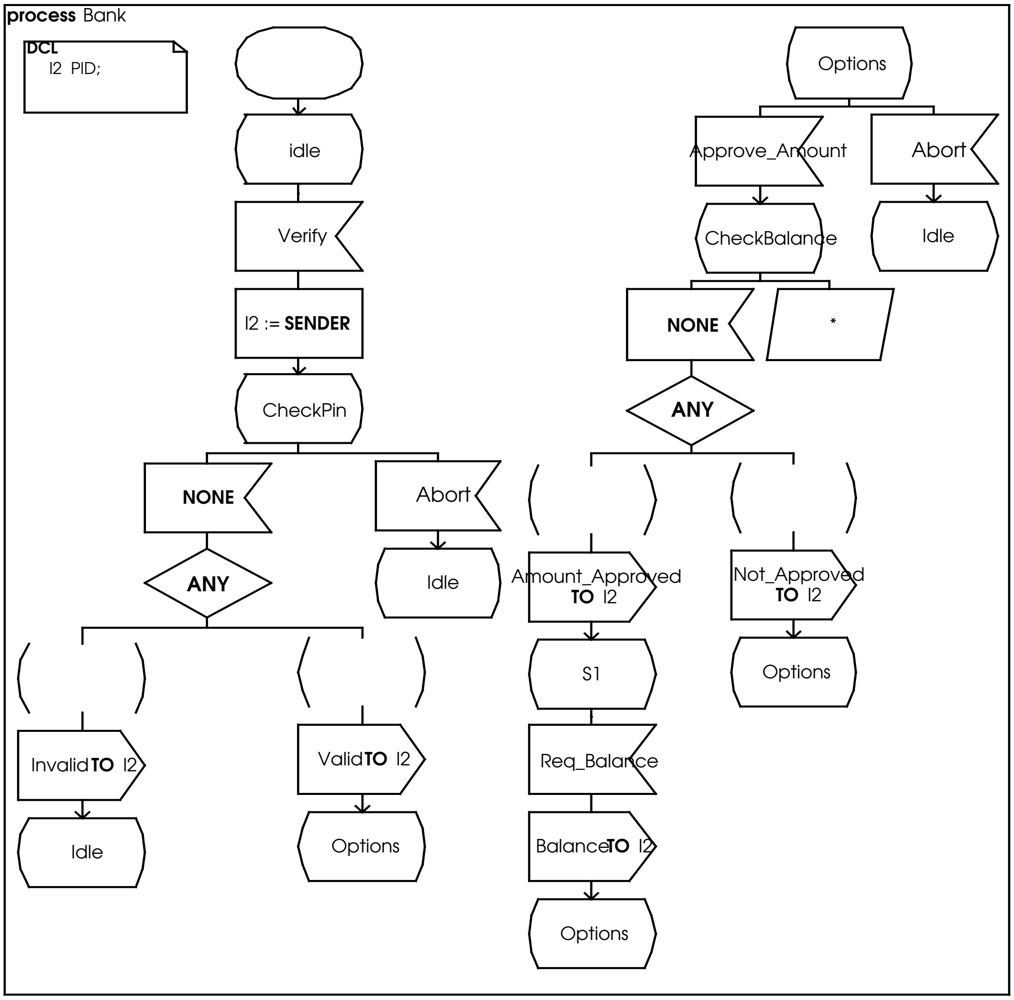

Figure 5. *Bank* process.

New Function to Add

Now, assume we have a new function, *GetBalance*, to add to our system. This new function (new behavior) is described with the MSC given in Figure 6. The function *GetBalance* involves two existing process instances, *ATM* and *Bank*. There is no need for a new process to be introduced. However, signals, such as *GetBalance*, *ReqBalance*, are new.

In the MSC *GetBalance*, we have used a global condition, *Options*, to indicate where the new behavior has to be added in the existing SDL processes. Notice the style of the given MSC is very important. The conditions are important for merging the new behaviors with the old ones. The user should know where the new behaviors have to be added. However, conditions do not have to be global, they can be just local.

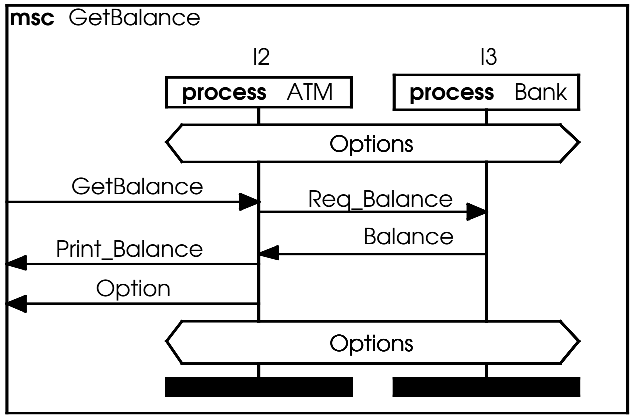

Figure 6. GetBalance function specification with MSC.

Obviously, we would like to preserve the existing functions, such as *Withdraw*. In other words, we would like to add this new feature without disturbing existing ones. The user should be able to use the *Withdraw* function for instance, without any change. Therefore, we would like to design a new specification that will be an extension of the original one.

Adding the MSC to the SDL specification

The first step in our approach is to complete the architecture. This consists on adding newly required channels or enriching existing channels to handle new signals. If new processes are described in the MSC, then the user has to specify where the processes fit in the architecture of the system. In the case of our working example, we have the processes in the MSC are all present and defined in the SDL specification. However, we have new signals to take into account and these signals are added to the existing channels depending on their source and destination. The resulting architecture (only the block *ATM_Block*) is shown in Figure 7.

The second step is to generate the SDL processes corresponding to the instance processes in the new MSC using *MSC2SDL* tool [7, 8]. Here, we take into account the enriched architecture, since it is an important parameter for the SDL processes generation algorithm. The generated SDL processes are given in Figure 8.

Third step is to merge each newly generated process with their corresponding process in the original specification. For each pair of processes (the original and newly generated one), the condition (multiple local conditions) specified in the MSC represent the SDL state where the new behavior has to be merged. If the input signal is not already explicitly handled in this state, then we add a new branch and the new behavior is defined in this new branch. If the input signal is already explicitly handled, then we check further for common behavior (output sequence) for this input. If the responds of the processes for this common signal are different, then the new behavior cannot be added to the old one without introducing a new non-determinism, which will be violating the behavior preservation restriction. When the responds are the same, we merge these two transitions into one, and continue the process with the next states in each process. We repeat this processing until all the new states in the new behavior are taken into account or a violation of the behavior preservation restriction is reached. The application of this last step to our example leads to the *ATM* and *Bank* processes as shown in Figure 9 and Figure 10, respectively.

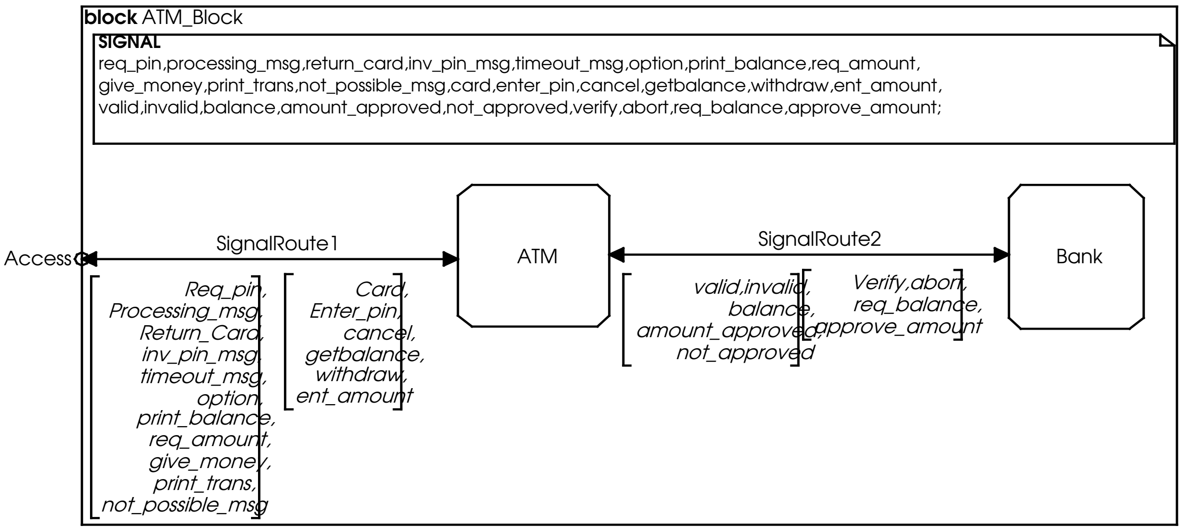

Figure 7. Enriched SDL architecture (*ATM_block*).

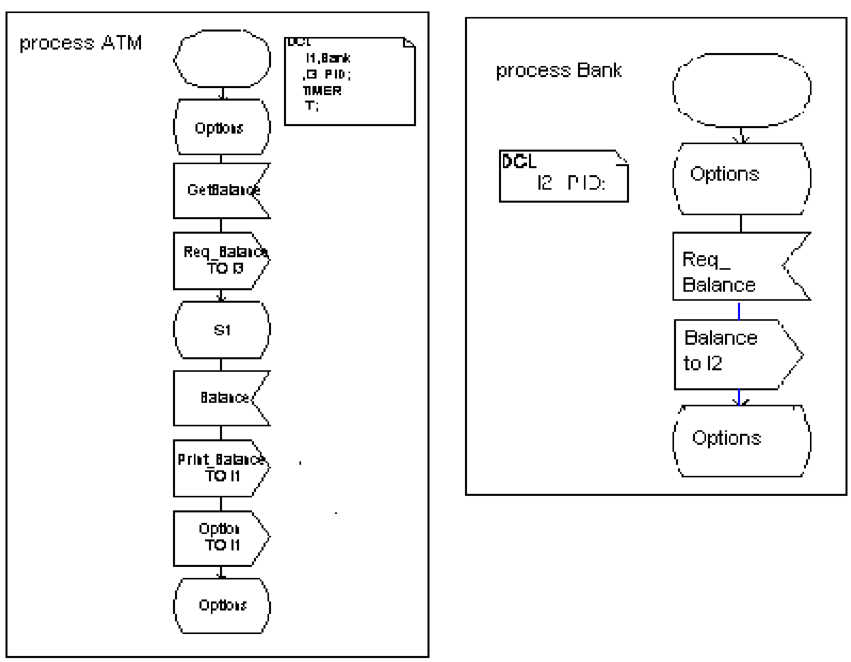

Figure 8. New behaviors in SDL.

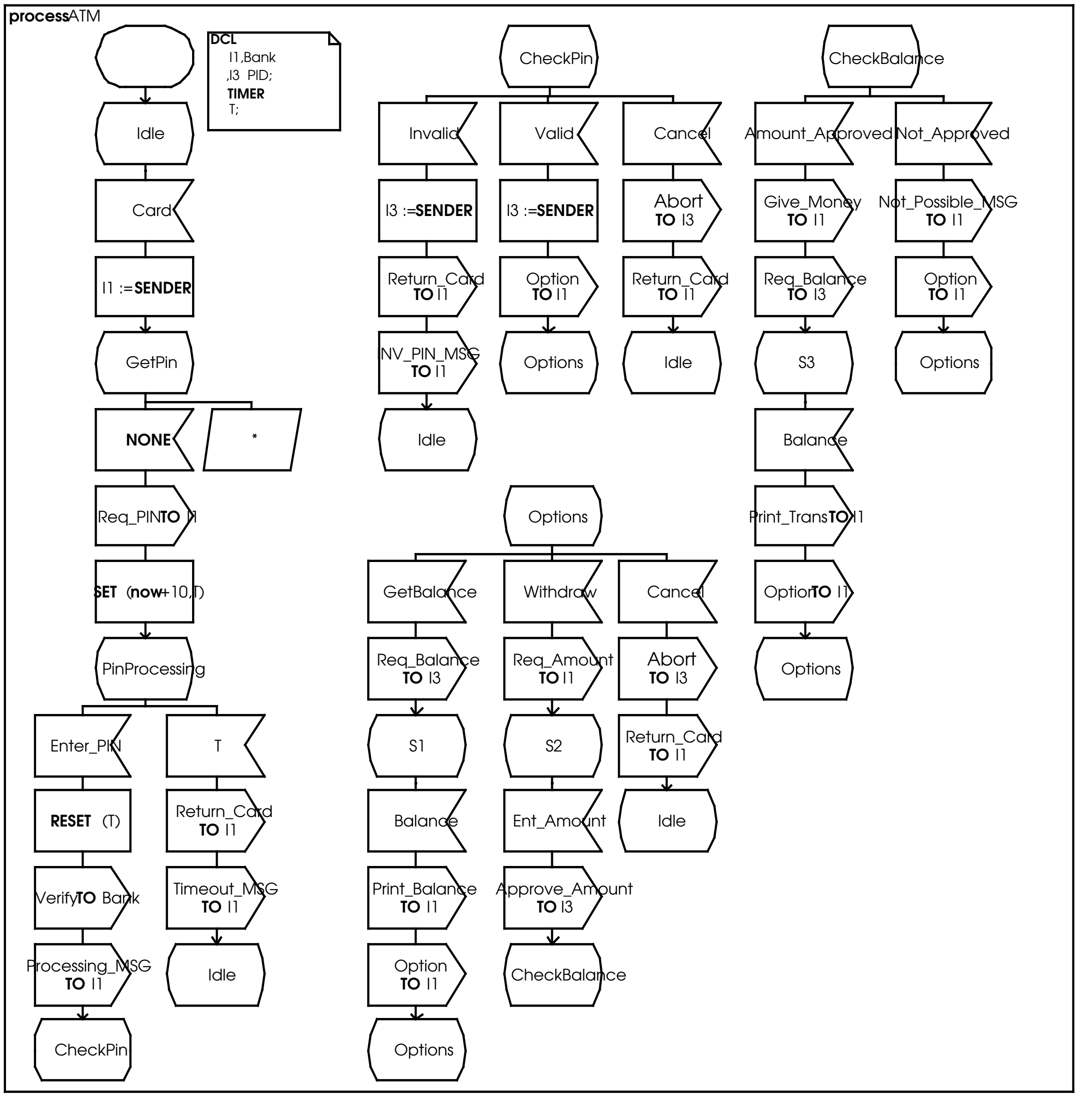

Figure 9. Enriched *ATM* process.

The process *ATM* given in Figure 9 is an extension of process *ATM* given in Figure 4. It does all what the initial ATM process does without any new non-determinism. Moreover, it provides a new function that can be used by the process environment. The new process *Bank* is also an extension of the process *Bank* given in Figure 5. We conclude that the new *ATM system* is an extension of the initial one. It provides more services to the users.

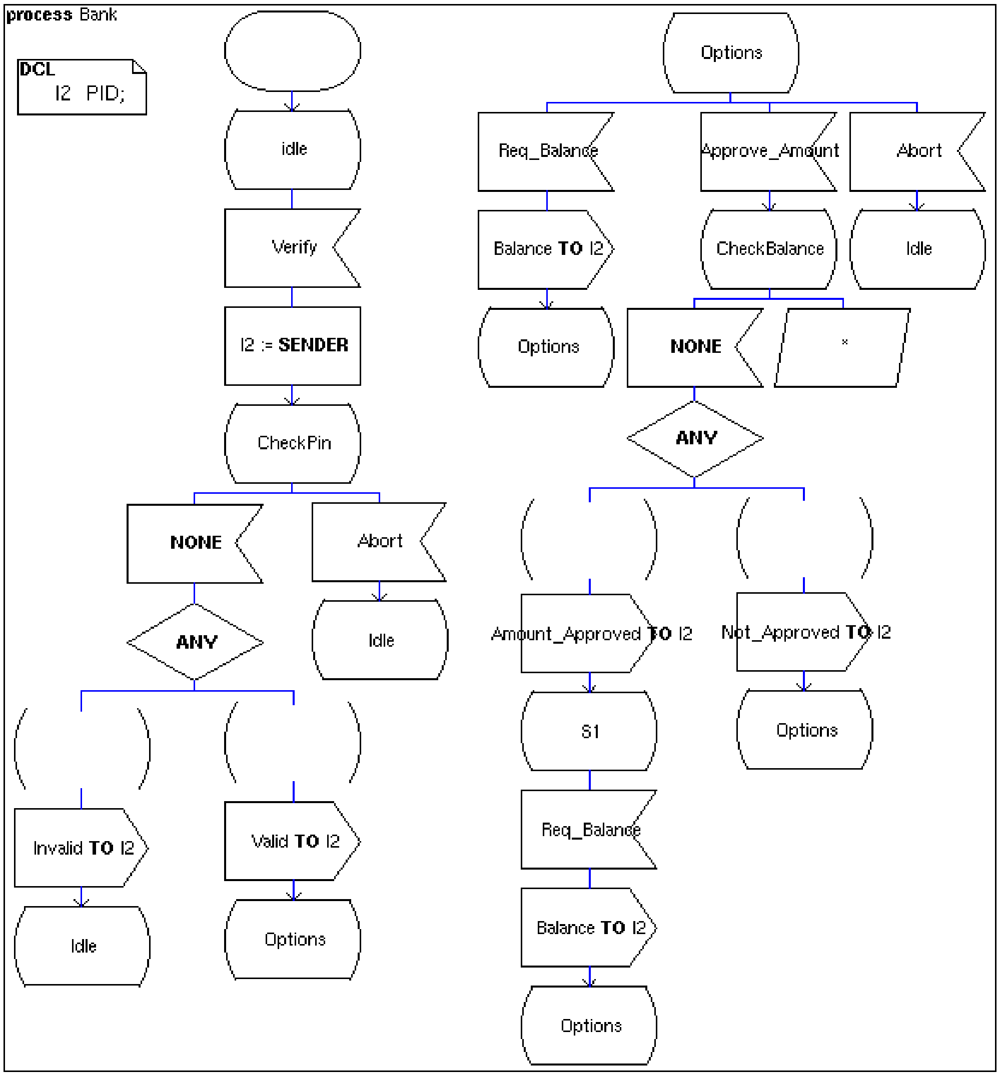

Figure 10. Enriched *Bank* process.

4. Conclusion

We have introduced a new formal relation for comparing SDL specifications. This relation has been defined for the purpose of specification design and maintenance. For other purposes, such as testing, other conformance relations could be more suitable. The extension relation we have introduced here combines concepts from the extension relation in [5], simulation relations and the specialization concepts already used informally in SDL [9, 10, 11, 12]. However, the comparison with existing relations is not straightforward, because of the *save* construct and the way SDL processes handle unexpected inputs. We are planning to investigate further this relation and its properties as well as the study of other formal relations.

We have described our SDL specification enrichment approach through an example. In the example MSC specification of Figure 6, the information such as instance names and condition names are important for the merging procedure. The new MSC has been specified with some knowledge, such as state names, of the existing SDL specification. As future work, we intend to investigate the possibility of using MSCs with less know knowledge about the existing SDL specification.

In this paper, we have focused on the behavioral enrichment. There exist other kinds of enrichments such as adding new components to the system and the combination of new behaviors and new components. There is no doubt that for enriching the architecture with new components (processes or blocks) the interaction with the developer is required. The block to which the new component belongs and the number of new communication channels between this component and the other ones can be decided by the developer only. As part of our future work, we plan to investigate more formally these types of enrichments and the related issues.

In this paper, we have considered the enrichment of an SDL specification with a new MSC. We have used the *MSC2SDL* tool to translate the new behaviors from MSC to SDL and then merge them with the corresponding ones in the original SDL specification. We propose to use this approach in the case where no MSC specification is available for the original SDL specification. However, if an MSC specification of the system exists, we can enrich the MSC specification and the SDL architecture first and then generate the new SDL specification using *MSC2SDL*. MSC language is more suitable for incremental design than SDL. For instance, if we consider the HMSC in Figure 11, we can see that this ATM specification can be extended easily with new functions with simple addition of alternatives to "*ProcessOption*". (Notice that the HMSC in Figure 11 follows ObjectGeode [3] format and not the standard syntax for HMSC.) However, this kind of additions does not always lead to an extension as defined in this paper, especially if two or more alternatives have some initial behavior in common.

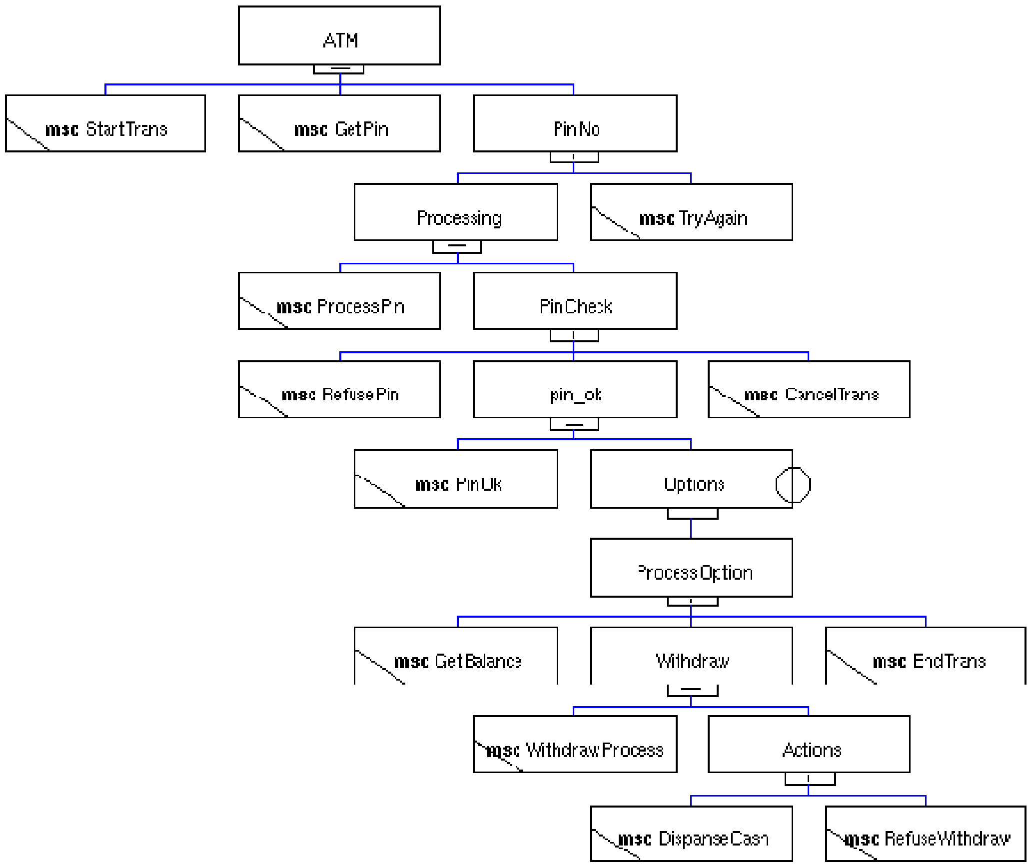

Figure 11. Enriching MSCs.

References

1. ITU-T, “Recommendation Z.120 - Message Sequence Charts (MSC)”, 1996.

2. ITU-T, “Recommendation Z.100-Specification and Description Language (SDL)”, 1996.

3. ObjectGeode, Verilog, Toulouse, France, 1996.

4. SDT Tau, Telelogic, Sweden, 1996.

5. E. Brinksma, G. Scollo and S Steenbergen, “LOTOS Specifications, their implementations and their tests”, Proceedings of Protocol Specification Testing and Verification (PSTV) 86, B. Sarikaya and G. Bochmann (eds), Montréal, Canada, June 1986.

6. R. De Nicola, “Extensional Equivalences for Transition Systems”, Acta Informatica, 24, 1987, pp. 211-237.

7. G. Robert, F. Khendek and P. Grogono, “Deriving an SDL Specification with a Given Architecture from a Set of MSCs”, in A. Cavalli and A. Sarma (eds.), SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the eight SDL Forum, Evry, France, Sept. 22 - 26, 1997.

8. M. M. Abdalla, F. Khendek, and G. Butler, “New Results on Deriving SDL specifications from MSCs”, Proceeedings of the SDL Forum’99, R. Dssouli, G.v. Bochmann, Y. Lahav (eds), Montréal, Canada, June 22-25, 1999.

9. O. Faergemand, “Stepwise Production of an SDL Description”, Proceedings of Formal Description Techniques (FORTE) III, Madrid, Spain, 1990.

10. J. Ellsberger, D. Hogrefe and A. Sarma, “SDL: Formal Object Oriented Language for Communicating Systems”, Prentice-Hall, 1997.

11. B. Moller-Pedersen, O. Haugen and D. Beslnes, “Introduction to Object-Oriented SDL and to Elements of corresponding Method”, Tutorial, Formal Description Techniques (FORTE) III, Madrid, Spain, 1990.

12. ITU-T, Supplement 1 to Recommendation Z.100 [Z.100 suppl.1] – SDL + methodology, 1997.

13. F. Khendek and G. v. Bochmann, “Merging Behavior Specifications”, Journal of Formal Methods in System Design, Vol. 3, pp. 259-294, 1995.

14. H. Ichikawa, K. Yamanaka and J. Kato, “Incremental Specification in LOTOS”, Proceedings of Protocol Specification Testing and Verification (PSTV) X, Ottawa, Canada, 1990.