MESA: Support for Scenario-Based Design of Concurrent Systems *

Hanêne Ben-Abdallah[1] and Stefan Leue[2]

[1] Faculté des Sciences Economiques et de Gestion, Université de Sfax
Sfax, Tunisia, hanene@swen.uwaterloo.ca
[2] Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada, sleue@swen.uwaterloo.ca

Abstract. The latest ITU-T standard syntax of Message Sequence Charts (MSCs) [16] offers several operators to compose MSCs in a hierarchical, iterating, and nondeterministic way. However, current tools operate on MSCs that describe finite, deterministic behavior. In this paper, we describe the architecture and the partial implementation of MESA, an MSC-based tool that supports early phases of the software development cycle. The main functionalities of MESA are: an environment for the composition of system models through MSCs, syntactic and model-based analysis of an MSC model, and resolution of resource related underspecifications in an MSC model.

1 Introduction

Message Sequence Charts (MSCs) have been extensively used in the development of telecommunication and reactive systems. They have already been adopted within several software engineering methodologies and tools, e.g., [13], [8], [17], [22], [7], [2], and [3]. MSCs are used to document system requirements that guide the system design [22], describe test scenarios (e.g., [17, 7]), express system properties that are verified against SDL specifications [2], visualize sample behavior of a simulated system specification [22, 2, 12], capture early life-cycle requirements [3], and to express legacy specifications in an intermediate representation that helps in software maintenance and reengineering [13].

In this paper we propose the architecture for an MSC-based tool for the requirements and design phases of the life-cycle of reactive systems. In addition, we illustrate how a portion of this architecture has been implemented in the *M*essage Sequence Chart *E*ditor, *S*imulator and *A*nalyzer (MESA) tool. The presented tool has several motivations. One is to serve as an integration platform for various tools, which gives software engineers an access to a wider range of design and analysis techniques that can be more effective due to certain customizations. The integration is facilitated through the standardized syntax of

* This work was partly supported by the Information Technology Research Centre of the Province of Ontario and by the National Science and Engineering Research Council of Canada. ObjecTime Limited provided further support.

MSCs by the ITU-T in Recommendation Z.120 [16]. In other words, we view MESA as a front-end to various methods and tools, and hence code synthesis from MSC specifications is an essential objective. We currently envisage synthesis of SDL [14], ROOM [22] and Promela [11], all of which enjoy support by mature, industrial strength CASE tools.

A second motivation for MESA is to extend the usage of MSCs to the requirements specification and design phases. Current tools that use MSCs operate on MSC specifications that describe finite and deterministic system behavior. However, in its recent extension, called high-level MSCs [15], the MSC language offers modular and hierarchical operators to describe parallel, sequential, iterating, and non-deterministic execution of basic MSCs. These operators facilitate the specification of large-scale systems. In addition, MSCs offer essential constructs in a *requirements* language for reactive systems, e.g., distinction between the *system* and its *environment*, communication exchanges, internal actions, and timers along a formal semantics [16, 18]. As a design language, the notion of *processes* in MSCs along the composition operators can be used to reflect a software architecture. However, iteration and nondeterminism in MSCs require additional, explicit information, e.g., underlying network architecture and interprocess synchronization to resolve nondeterminism [5]. For this, an MSC-based tool for the design of reactive systems must offer analysis techniques to detect instances where such additional information is required and prompts the user for it.

In addition to the above type of design-related analysis, a third motivation for the tool is to provide analysis for high-level MSC models. Currently, tools that support analysis of MSCs operate only on basic MSCs. In particular, the MESA tool offers an extension of MSCs with real-time information and supports timing analysis for high-level MSCs. The time extension is based on currently evolving propositions ([16, 3].)

Another motivation for our tool is that we believe that assertional reasoning is crucial for any realistic analysis of a system model. For this, we benefit from the integration features in MESA and use existing model-checkers, more specifically, by synthesizing code that serves as an input to a model checker, e.g., Promela code for the XSPIN tool [12].

Paper organization. Section 2 discusses the suitability of MSCs for requirements specification and design, while Section 3 reviews current usages of MSCs in software engineering tools. Section 4 illustrates the usage of MSCs based on an automatic teller machine (ATM) example. Section 5 presents the architecture of MESA as an MSC-based tool for the requirements and design phases. Section 6 describes the currently implemented version of MESA and its application to the ATM example. Section 7 summarizes the paper and outlines future research directions.

2 Role of MSC-based Tools in the Software Lifecycle

The standard syntax of MSCs is defined by the ITU-T Recommendation Z.120 [16]. A basic MSC (bMSC) essentially consists of a set of processes (called instances in Z.120) that run in parallel and exchange messages in a one-to-one, asynchronous fashion. In addition to exchanging messages, processes can individually execute internal *actions*, use *timers* to express timing constraints, create and terminate process instances. The standard extension of the MSC language, called *High-Level MSCs* (hMSCs) [16], provides for operators to connect *basic* MSCs to describe parallel, sequential, iterating, and non-deterministic execution of basic MSCs. In addition, hMSCs can describe a system in a hierarchical fashion by combining hMSCs within an hMSC.

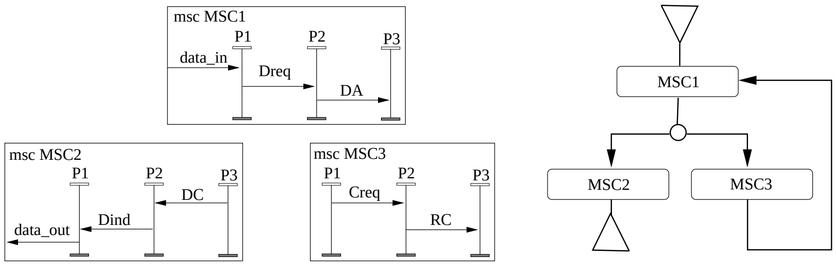

Fig. 1. MSC specification example: basic MSCs (left) and high-level MSC (right)

As an example of an MSC specification, consider Figure 1 which describes a simple connection establishment protocol in a telecommunication system. Process `P1` is a service provider, `P2` is a local and `P3` is a remote protocol machine. The iterating branch describes a repeated request to establish the connection. The non-iterating branch describes a successful connection establishment. The semantics of an MSC essentially consists of sequences (or traces) of messages that are sent and received among the concurrent processes in the MSC. The order of communication events (i.e. message sent or received) in a trace is deduced from the visual flow of control within each process in the MSC along with a causal dependency between the event of sending and receiving a message [18].

Message Sequence Charts offer several advantages to the requirements and design phases of the development of reactive systems. One is the intuitive, graphical notation of MSCs which helps a designer to visualize the system's structure and interfaces. In addition, as a *requirements* specification language an advantage of MSCs is the level of abstraction they offer by merely describing the message flow between processes (which is the core of reactive systems) and abstracting out process behavior. This is to be contrasted to other specification techniques, e.g., SDL and ROOM, which explicitly specify the process behavior and leave the message flow implicit. Another advantage of MSCs as a requirements language is the distinction between the actions of the system and its environment: MSCs visually distinguish between the actions the system produces or initiates from those produced by the environment, which facilitates the identification of the interface between the two system components. For example, in Figure 1 the environment sends a message of type `data_in` to the system through the process

P1; the system sends a message of type `data_out` to the environment through the process P1. As a *design language*, MSCs are suitable through their notions of processes and composition operators which facilitate the description of the system's software architecture.

Due to the focussed expressiveness of MSCs, it is unrealistic to build a CASE tool exclusively on MSCs. A tool based on MSCs should be a front-end to other CASE tools, supporting early life-cycle requirements capture and validation capabilities (see also [3]). In addition, an MSC-based tool should support a number of functionalities including:

- *Analysis* of MSC specifications for "*things that can go wrong*". The graphical appeal and perceived clarity of MSC specifications contrasts limits in their expressiveness, which in turn may lead to ambiguities due to underspecifications. For example, in the presence of non-determinism and iteration in an hMSC, explicit information is required about inter-process synchronization and the underlying network architecture and queuing strategies [5]. The lack of this information may lead to discrepancies between an MSC specification and its interpretation and thus any potential implementation [5]. Thus, it is essential to support analysis mechanisms that detect such ambiguities and suggest to the designer possible extensions to resolve them. Other analyses include analysis of the consistency of timing constraints attached to an MSC, and semantic analysis that checks safety and liveness properties.
- *Synthesis* of code skeletons in full-fledged specification notations such as SDL, ROOM or Promela, and *testing* notations such as TTCN when testing support is crucial. Code synthesis allows the MSC tool to be integrated with other tools that provide additional functionalities e.g. model-based analysis.
- Means to *simulate* the execution of MSC specifications.
- A GUI-based *editor* to manipulate MSC specifications.

Based on the above tool requirements, we have designed and partly implemented a tool called MESA (*M*essage Sequence Charts *E*ditor, *S*imulator and *A*nalyzer) to support software design for concurrent systems at early life-cycle stages.

3 MSCs in Software Engineering Tools

In our review of current tools (c.f. [4]) that use MSCs, we examined the following set of requirements we find important in the requirements and design phases as previously described: 1) The use of MSCs in the description of reactive systems makes constructs to support *branching* and *iterating* indispensable. 2) Overall compliance with a syntactic convention like Recommendation *Z.120* or the UML [8] notation is desirable. 3) While a translation from MSCs into a different formalism, e.g., for analysis purposes may be necessary, we require that as little *semantic bias* as possible be included. In particular, we criticise the interpretation of MSCs based on SDL, as sometimes proposed, because of SDL's heavily constraining message passing semantics. 4) In a notation that benefits from graphical appeal and visual allusion to such an extent as MSCs, it is mandatory to have semantic assumptions explicitly represented in the specification. Allowing implicit semantic assumptions would defeat the purpose of using MSCs during the requirements and design phases where precise specifications are vital. We call this requirement the "*what-you-see-is-what-you-get*" (WYSIWYG) requirement w.r.t. the semantics given to MSCs as described by their visual representation. 5) A requirements tool needs to provide for means to check the *consistency* of the requirements specified. 6) Executing or simulating MSCs can greatly enhance the debugging of a specification, and thus *simulation* should be provided by an MSC-based tool.

We have analyzed a sizable subset of the tools that support MSC specifications (c.f. [4]). The set of analyzed tools includes:

- The *ObjecTime* toolset by ObjecTime Limited which supports editing of basic MSCs to document requirements on the communication behaviour of an ObjecTime model [22] and to visualize execution traces of an ObjecTime model.
- The SDL based tools *GEODE* [2] by Verilog SA and the *SDT* [1] by Telelogic AB both support editing of basic MSCs as well as a requirements validation check that answers whether at least one execution of the SDL system corresponds to a given bMSC.
- The *MSC Analyzer/POGA* tool by Bell Labs [3] is centered around bMSCs and provides means to analyze an MSC specification syntactically for timing constraints, and analyze it with respect to potential discrepancies between the perceived and the implied event ordering in an MSC. This tool supports the editing of hMSCs, but analysis is focussed on only basic MSCs.
- The *SDE* [13] and *MuSiC*++ [20] developed by NTT are closely related tools centered around SDL and providing for MSC editing, analysis and code synthesis. Analyses include an inconsistency check that is based on deadlock detection in bMSCs, and code synthesis that produces SDL code from bMSCs.

From our review of the above tools, we found out that none of them satisfactorily meets the requirements we outlined earlier. In particular, Only the *MSC Analyzer/POGA* tool supports Z.120-style MSC composition. Users are allowed to identify a simple path in the hMSC graph, which is then composed by concatenation to form a large bMSC that in turn is analyzed. Furthermore, while the *SDE* and *MuSiC*++ tools provide for a variety of analysis functions, these tools suffer from a heavy bias towards an assumed SDL semantics. For instance, an MSC could be flagged as deadlocking even though it will not deadlock unless one assumes the rather constraining SDL semantics. The SDL-based semantic assumption contradicts our *WYSIWYG* requirement.

4 MSC Specification Example: an ATM System

In the remainder of this paper, we will use an Automated Teller Machine (ATM) example[1] to illustrate the functionalities of Mesa. Figures 2, 3 and 4 illustrate

[1] A variant of this example was first presented in [6].

the MSC-based specification of the ATM example. All diagrams were generated through the editor component of MESA.

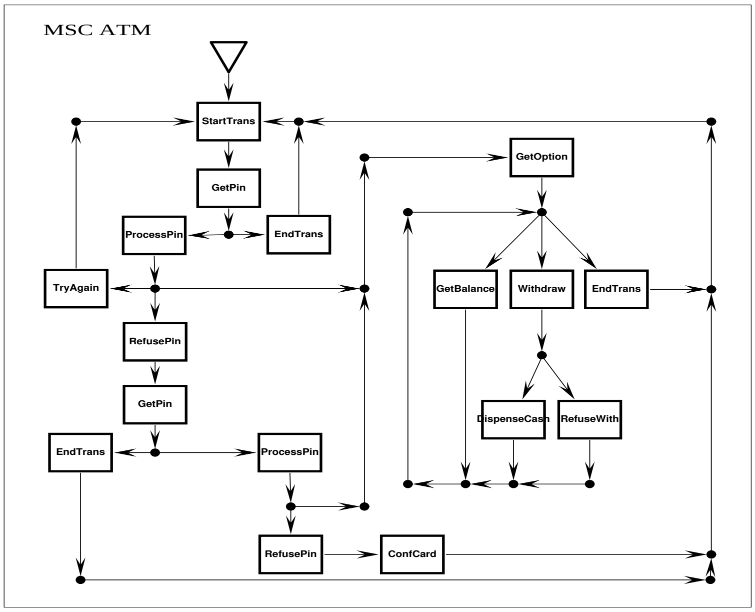

Fig. 2. hMSC for ATM System

Due to space limitations, we briefly review the example; for more detail see [6]. The ATM system consists of three concurrent components: the system's user interface that communicates with potential customers represented by the `User` process, the ATM controller software which is represented by the `ATM` process, and a host computer in a central bank office that is represented by the `Bank` process. Each one of the bMSCs in Figures 3 and 4 represents a scenario or 'use-case' of the system. The hMSC graph in Figure 2 specifies a successor relationship between these scenarios.

5 Architecture of an MSC Requirements and Design Tool

Figure 5 presents a data flow diagram-like view of the architecture of MESA. The four main functions of MESA(editing, syntactic analysis, model-based analysis, and code synthesis) are accessed through the GUI-based editor for hMSCs and bMSCs. Figure 6 shows the hMSC editor displaying the hMSC graph of the ATM example. Double clicking with the mouse on one of the boxes that represents a bMSC opens an editor window for the bMSC that is linked to this box. Figure 7 shows the bMSC editor windows for the `DispenseCash` and `Withdraw` bMSCs.

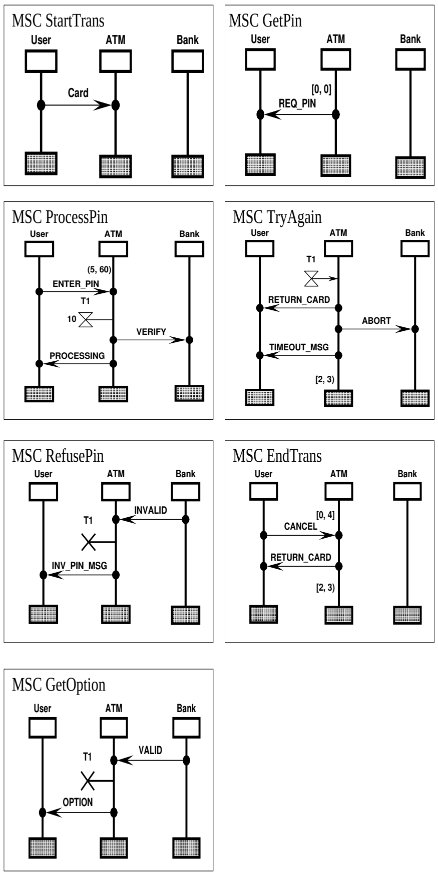

Fig. 3. Part 1 of ATM System (bMSCs)

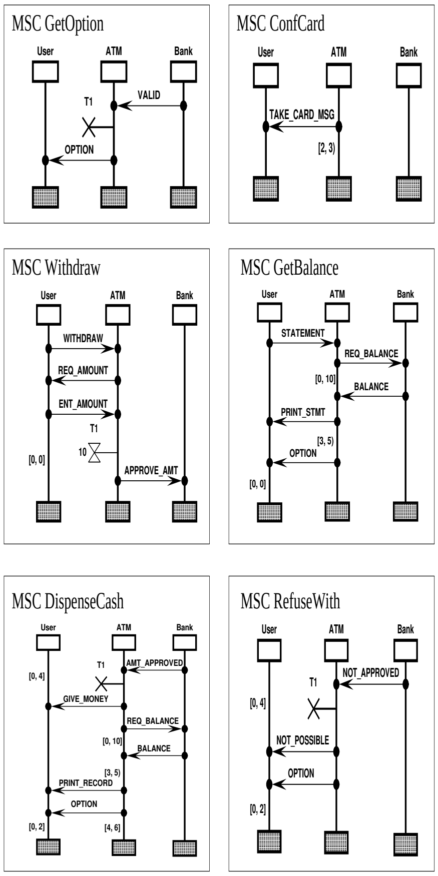

Fig. 4. Part 2 of ATM System (bMSCs)

5.1 Editing

The Mesa editor allows the user to textually input, draw and manipulate both the hMSC and bMSC components of an MSC specification. In addition, it allows the user to load and store MSC specifications.

Syntactic Checks. There were two design goals for the editing component within Mesa. First, users should be guided in following the graphical syntax of Z.120 [16]. Second, we were interested in leaving users the freedom to choose compliance with the Z.120 standard to some extent.

Slope of message arrows: Z.120 requires message arrows to be either downwards sloping or horizontal. In [5] we showed that this is a sufficient syntactic condition to avoid deadlocks in an MSC specification. The Mesa editor optionally enforces this rule when drawing message arrows.

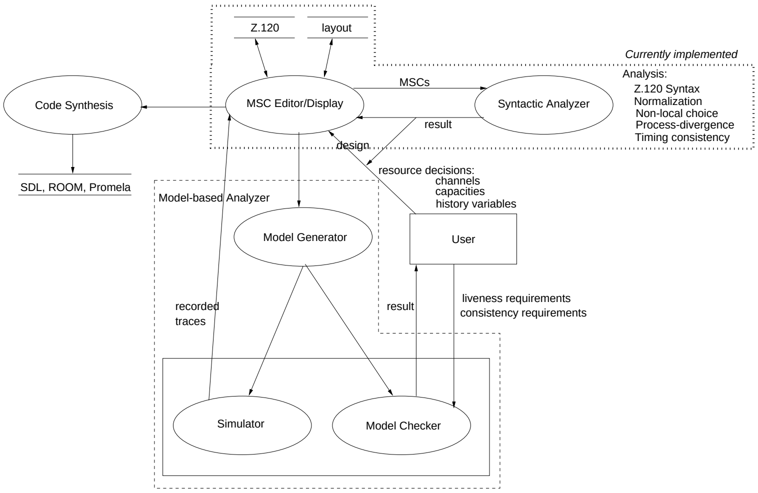

Fig. 5. Architecture of MESA

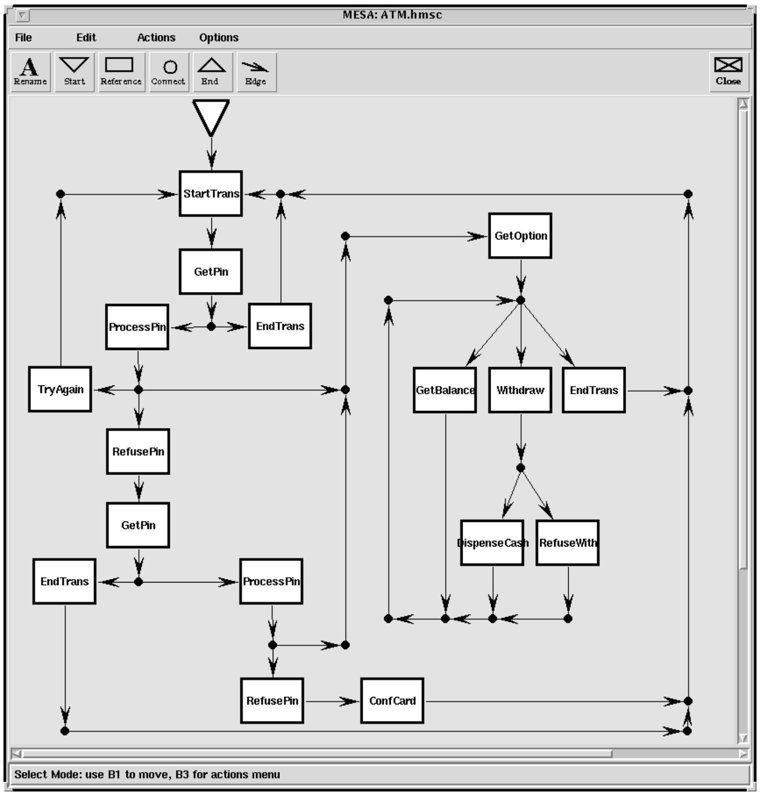

Fig. 6. Snapshot of the hMSC editor

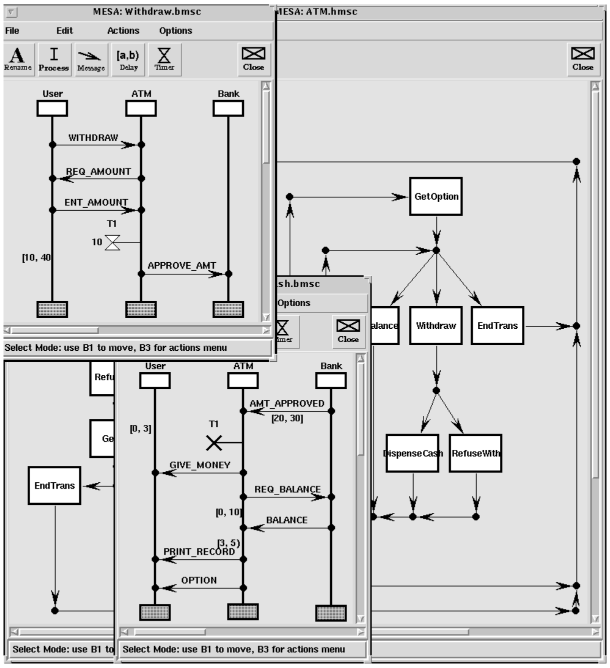

Fig. 7. hMSC and bMSC editors

Graphical well-formedness and compliance with Z.120: MESA checks *bMSCs* for compliance with the following constraints: 1. Each process must be given a unique name within the scope of a bMSC; 2. each message type must always be sent and consumed by the same pair of processes within an MSC specification; 3. a message arrow may not be directed upwards; and 4. two messages may not originate from the exact spot on a process axes to avoid ambiguities in the interpretation of the event ordering. For *hMSCs*, MESA checks the following consistency requirements: 1. All references to bMSCs must be defined; 2. all referenced bMSCs must be legal according to the above bMSC rules; 3. there must be exactly one start node; 4. the hMSC graph must be connected; and 5. the list of process names for each bMSC that is referenced in an hMSC must be identical in all bMSCs. In addition, MESA verifies whether an MSC specification is normalized, i.e. that after a branching in the hMSC graph no two bMSCs have an identical message exchange prefix. Normalization is needed for some analyses [5].

Input and Output Formats. MSC specifications can be stored in three different data formats:

– *Strict Z.120 textual format* [16]. Compliance with the standard textual syntax facilitates sharing of MSC specifications amongst different CASE tools.

- *Extended Z.120 textual fomat.* The strict textual representation of MSCs is insufficient to completely reflect the information content of bMSCs as we use them. Therefore, the syntactic representation that we use extends the strict Z.120 syntax as follows.
  1. *Layout information.* This information is essential to reproduce chart layouts that the user has previously chosen. MESA extends the Z.120 textual syntax with formatting information that is bracketed by comment symbols '`/*`' and '`*/`'. The layout information consists of the coordinates of the various MSC components relative to the upper left corner of the MESA editing canvas, which means that this information may be meaningless for other tools. Figure 8 shows the automatically generated textual Z.120 extended with layout representation for the bMSC `Withdraw` of Figure 4. A similar technique is chosen to represent layout information for hMSC graphs.
  2. *Timing information.* MESA represents two types of timing information in bMSCs: the Z.120 timer-based real-time constraints, and our suggested delay interval-based real-time constraints [6]. Accordingly, MESA extends the textual Z.120 syntax to include the suggested timing constraints as follows:
     - Delay intervals along *process axes*: We introduce a clause `delay [l, u]`[2] in between two consecutive events within a process, c.f. the `delay[0, 0]` clause in Figure 8.
     - Delay intervals along *message arrows*: We extend the Z.120 message clauses with a delay interval. For example, consider the sending of a message of type `A` to a process `P`, which is represented by the clause `out A to P;`. To add the timing constraint that this message arrow has been labeled by a `delay [l, u]` clause, we extend this clause to `out A to P delay [l, u];`.
- *Encapsulated Postscript.* To support the use of MSCs in software requirements and design documentation, MESA generates encapsulated postscript code of hMSC graphs and bMSCs.

Graphical Editor. The GUI-based editor (c.f. Figures 6 and 7) provides an icon-based drawing palette with the basic components of the bMSC and hMSC languages. The GUI-based editor is implemented with an object-oriented interface. One mouse click on a drawn object shows a menu of actions associated with the object.

5.2 Syntactic Property Analysis

It is often less expensive to verify properties of a specification syntactically as opposed to analyzing the specification's model. Currently, MESA implements three types of properties that can be efficiently checked syntactically: *process divergence*, *non-local branching choice* and *timing consistency* (c.f. [5]).

[2] `l` and `u` denote numeric values representing the *lower* and the *upper* bound for the message delivery delay, respectively.

```
msc Withdraw;
inst  User, ATM, Bank;
        instance User   /* x=72 length=280*/;
                out WITHDRAW to ATM     /* y=76 */;
                in REQ_AMOUNT from ATM  /* y=110 */;
                out ENT_AMOUNT to ATM   /* y=146 */;
                delay [0, 0];
        endinstance;
        instance ATM    /* x=190 length=280*/;
                in WITHDRAW from User   /* y=76 */;
                out REQ_AMOUNT to User  /* y=110 */;
                in ENT_AMOUNT from User /* y=146 */;
                set T1 (10)     /* y=187*/;
                out APPROVE_AMT to Bank /* y=217 */;
        endinstance;
        instance Bank   /* x=304 length=280*/;
                in APPROVE_AMT from ATM /* y=217 */;
        endinstance;
endmsc;
```

Fig. 8. Z.120 compliant textual syntax of the `Withdraw` bMSC generated by MESA

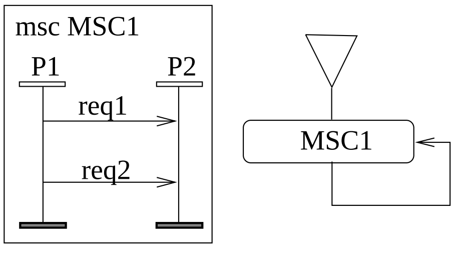

Fig. 9. MSC specification with process divergence

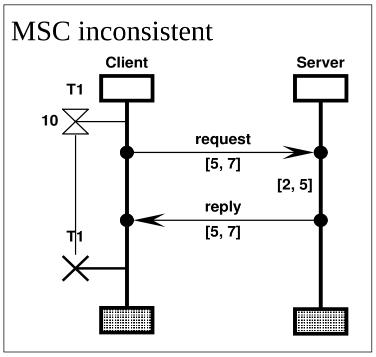

Fig. 10. Timing inconsistent bMSC

Process divergence. When processes iterate in an MSC specification, the asynchronous nature of communication can lead to *process divergence*: a system execution where one process sends a message an unbounded number of times ahead of the receiving process. Since an MSC specification makes no assumption about the speed of its processes, in the absence of a *hand-shake* mechanism, a sender process can run "faster" than a receiver process – possibly flooding the receiver with messages. As an example of process divergence, consider the MSC specification of Figure 9. One possible execution of this MSC specification is the infinite trace !`req1` !`req2` !`req1` !`req2` $\cdots$ which is the result of process `P1` sending messages without process `P2` receiving any one. To handle such a potential execution, the implementation must answer several questions: What is the network architecture between the processes `P1` and `P2`? Is there any queuing mechanism and protocol and if so, what is the capacity of the channels? How are multiple copies of a not-yet received message handled, are they overwritten or are they buffered? Regardless of the answers to the above questions, none of them is based on information explicitly described in the given MSC specification. In addition, different answers may result in different implementations. We view process divergence as potentially *unintended* behavior of the specification that must be detected and brought to the designer's attention. This allows the designer to decide either to modify the specification to resolve the problem (e.g., by adding explicit hand-shakes), or to postpone the problem to the implementation phase. We have syntactically characterized process divergence and developed an algorithm (now implemented in MESA) that runs in a time linear with the number of messages in an MSC specification [5]. The algorithm basically examines the bMSCs involved in a loop and verifies that the processes within the bMSCs communicate through a hand-shake.

Non-local branching. Figure 1 illustrates an example which describes a system where `MSC1` is followed by either `MSC2` or `MSC3`. At this level of abstraction, all current interpretations assume that all processes choose the same alternative flow of control so that the overall system behavior is described by one basic MSC at a time. As argued in [5], in terms of implementation of individual processes, such an assumption can however be non-trivial as it requires additional, dynamic information about which alternative other processes in the specification took. For example, consider the specification in Figure 1. Assume that, after executing the `Dreq` event, process `P1` is the first process to decide whether to go 'left', i.e., the next bMSC to execute is `MSC2`. In order to implement properly the semantics of choice, the processes `P2` and `P3` must be informed about `P1`'s decision so that they branch accordingly. However, neither the MSC semantics as presented in Annex B of Z.120 [16] nor hMSC graphs provide an explicit way to handle such an information exchange. MESA implements our algorithm to detect non-local branching choices and which executes in a time linear with the number of messages in an MSC specification [5]. The basic idea behind our algorithm is to examine the bMSCs involved in a choice and verify that they all have the same, unique process which sends the first event. In case an MSC specification contains a non-local branching choice, our syntactic analysis produces the bMSCs that are involved in the non-local branching. This allows the user to resolve the choice by modifying the relevant bMSCs.

Timing Analysis. The usage of timing constraints may lead to specifications that have no timed execution, i.e. the specification is timing inconsistent [6]. Consider the bMSC in Figure 10. Obviously, the minimum time that passes from sending message `request` until receiving message `reply` when following the messages and the processing within process `Server` is at least 12 seconds. However, within process `Client` it is assumed that timer `T1`, which is set to 10 seconds right before sending `request`, is not expiring before `reply` is received. This means that the conjunction of all timing constraints is not satisfiable by any system execution. MESA implements our timing analysis algorithm for branching and iterating MSC specifications in [6] which extends work in [3, 9].

6 Using MESA in the Analysis of the ATM System

The automatic analysis of the ATM example shows that there are no syntactic anomalies and inconsistencies with Z.120 syntax in this specification. This analysis ensures that the subsequent analysis algorithms deliver meaningful results.

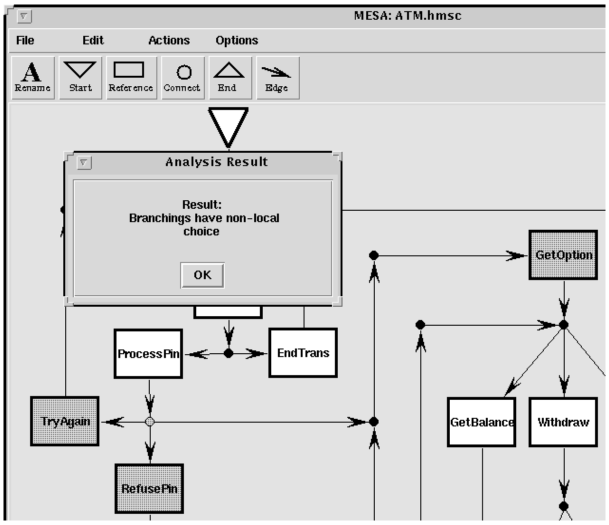

Fig. 11. MESA reporting *Non-local Choice*

Non-local Choice. Figure 11 illustrates how MESA reports the presence of a non-local choice situation in the ATM System. The non-local choice lies in the hMSC branching point that follows the bMSC `ProcessPin`: in `RefusePin` and

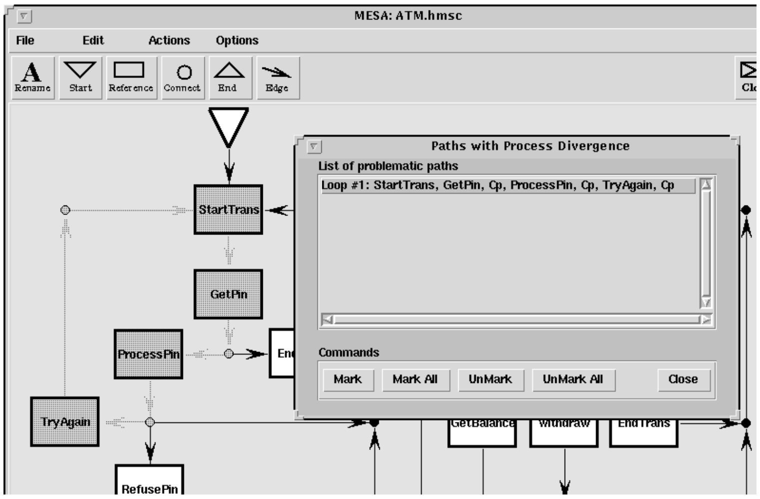

Fig. 12. MESA reporting *Process Divergence*

GetOption it is the Bank process which carries out the first action, i.e. sending of INVALID or VALID, respectively; whereas in TryAgain it is the ATM that sends the first message. The analysis reveals the danger that processes proceed in different directions at this branching point. MESA tags the respective parts of the hMSC graph which allows the user to localize the potential underspecification. The non-local choice situation here could be resolved in case it was not left up to the ATM process to send an ABORT message, but if the Bank was enabled to do so. The purpose of this analysis is to make the user aware of potential underspecifications, and to hint that synchronization mechanisms must be added, for instance, before code synthesis can proceed.

Process Divergence. The reporting of a Process Divergence situation in the ATM system is illustrated in Figure 12. The Process Divergence occurs in a loop in the hMSC that is formed by the bMSCs StartTrans, GetPin, ProcessPin and TryAgain. In this cycle, User and ATM exchange messages in both directions, while the Bank only receives messages. Depending on the speed of the processes, the ATM may be racing ahead of the Bank and sending a large number of VERIFY and ABORT messages to Bank. This indicates that in an implementation must carefully design the communication channel between ATM and Bank. A possible remedy of this Process Divergence situation is to increase the level of synchronization between the three processes. For example, the TryAgain scenario could be extended so that after sending the ABORT message to Bank the ATM process would only proceed after it had received a CONFIRM_ABORT message from Bank.

Timing Consistency. In the basic version shown in Figures 2, 3 and 4 all the timing assignments are consistent. But let us assume that we are more concerned about analyzing the timings in the example. Assume that we change the original specification of the ATM system as indicated in Figure 13: the transmission of `APPROVE_AMT` takes $[4, 7)$ seconds, the computations of `Bank` to determine that the amount cannot be approved take $[3, 5]$ seconds, and the transmission of the `NOT_APPROVED` message consumes $[4, 7)$ seconds. The question is now whether these new timing constraints are consistent with the remainder of the timing constraints in the system.

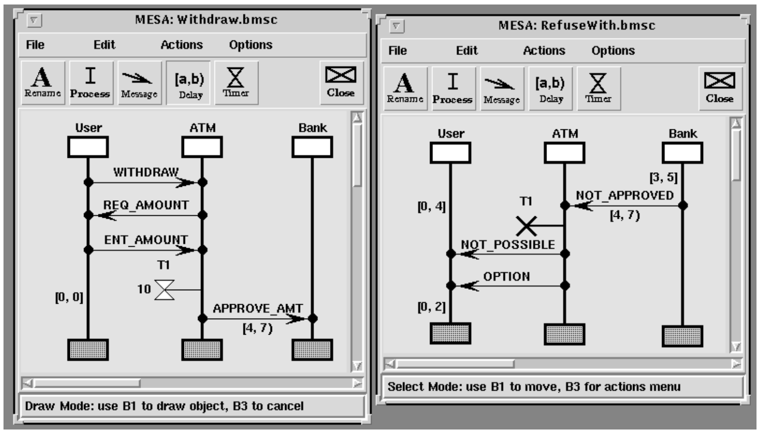

Fig. 13. Altered bMSCs in the ATM example

Timing analysis in MESA shows that these timing constraints are in conflict with the 10 second timer setting of `T1` in `Withdraw` and the pre-expiry reset of this timer in `RefuseWith`. Figure 14 illustrates how MESA reports the presence of a timing inconsistency by displaying a list of simple loop paths through the hMSC graph that have a timing inconsistency. MESA allows the user to tag some or all of the timing inconsistent loops in order to localize the timing inconsistencies. In Figure 14 we have tagged the loop # 4, which directly connects `Withdraw` and `RefuseWith`. For the ATM system, the timing analysis takes about 10 seconds of execution time on a Sun Sparc Ultra 1 –200 MHz system.

7 Conclusion

We have proposed an architecture of a tool for the requirements specification and design of reactive systems based on Message Sequence Charts. We have described the MESA toolset as a partial implementation of this architecture. MESA is designed to be a research testbed for a large variety of algorithms and methods developed around the MSC notation.

Our tool has some similarities with the *MSC Analyzer/POGA* tool [3]. These similarities concern aspects of data formats chosen (Z.120, Postscript), the timing analysis algorithm for *basic* MSCs, and probably aspects of the graphical

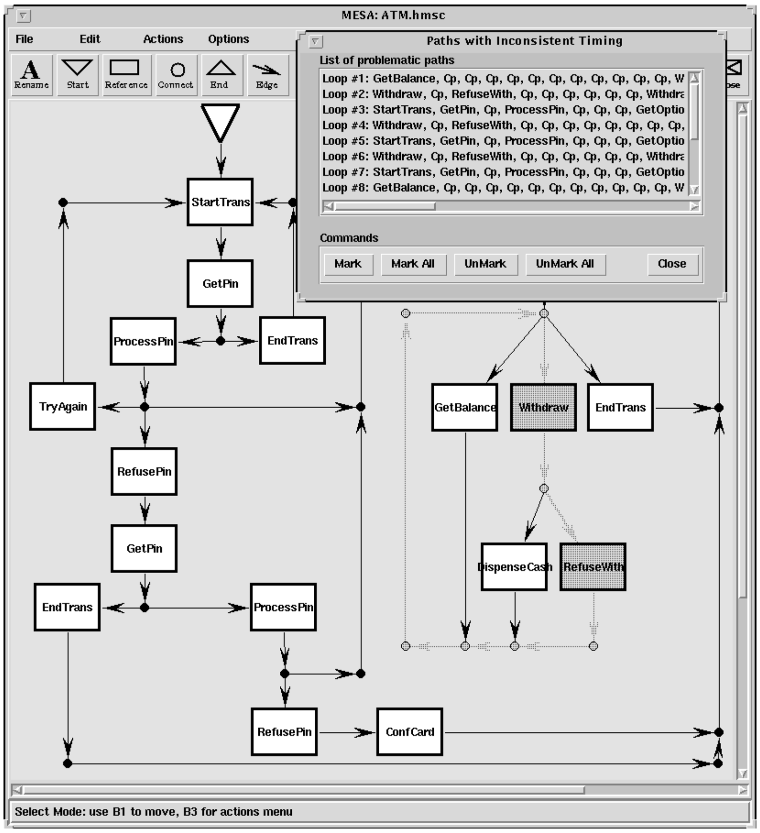

Fig. 14. MESA reporting timing inconsistency in the ATM System

user interface[3]. With respect to analysis algorithms MESA implements the following analyses that the MSC Analyzer/POGA tool does not offer: non-local choice, process divergence, and timing consistency for branching and iterating MSC specifications. In the MSC Analyzer/POGA tool it is necessary to perform manual unfoldings of the hMSC graphs in order to analyze cyclic MSC specifications. MESA performs an exhaustive search for all cyclic paths as well as a complete timing consistency analysis based on the theory discussed in [6].

Currently, we are preparing the public release of version 1.0 of MESA. It consists of those parts marked as "currently implemented" in Figure 5. At the time of writing we have spent approximately 6 man-months on code development for MESA, excluding basic research effort. The tool consists of approximately 15,000 lines of C++ and 3,000 lines of Tcl/Tk code. While the system is currently based on Unix Solaris operating system, we intend to port it to a number of

[3] The MSC Analyzer/POGA tool is not publicly available, hence we speculate graphical user interface similarities.

different platforms. Part of the documentation has been done using OMT [21] object modeling diagrams to accommodate the particular needs of the volatile University environment in which this system is being developed. As an "off-the-shelf" component we used the LEDA C++ library [10] for data structures like graphs and strings.

The effectiveness of the practical use of MESA hinges upon the reporting mechanism for analysis results. We are currently working on extending the *timing analysis* reporting such that not only the timing inconsistent loops in the hMSC are displayed, but also events involved in timing inconsistent loops together with the amount of timing inconsistency. To accommodate the *under-specification* of networking resources and branching synchronization mechanism we will allow the user to specify communication channels including capacities and history variables as first suggested in [19]. We are currently developing algorithms to synthesize SDL and ROOM models and we are implementing the synthesis of Promela code as suggested in [19]. Finally, to support *model analysis* and *simulation* capabilities we will pursue two routes: First, the translation into Promela together with the XSPIN model checking tool can accommodate for both features. Ultimately we would like to implement a generic simulator and model checker for MSC specifications based on MESA, which could be more efficient when based directly on the MSC objects.

Acknowledgements

The design of the user interface was based on preliminary work done by Tuan Ngo. Jennifer Hunt was instrumental in the design and implementation of the current version of MESA. She also suggested the extensions to the textual Z.120 syntax.

References

1. Telelogic AB. SDT. In G. von Bochmann, R. Dssouli, and O. Rafiq, editors, *Participant's Proceedings of the 8th International Conference on Formal Description Techniques FORTE'95, List of tools for demonstrations*, page 455, 1995.
2. B. Algayres, Y. Lejeune, F. Hugonment, and F. Hantz. The AVALON project: a validation environment for SDL/MSC descriptions. In O. Faergemand and A. Sarma, editors, *Proceedings of the 6th SDL Forum, SDL'93: Using Objects*, October 1993.
3. R. Alur, G. J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055*, pages 35–48. Springer Verlag, 1996.
4. H. Ben-Abdallah and S. Leue. Architecture of a requirements and design tool based on message sequence charts. Technical Report 96-13, Department of Electrical & Computer Engineering, University of Waterloo, October 1996.
5. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1217*, pages 259–274. Springer Verlag, 1997.

6. H. Ben-Abdallah and S. Leue. Timing constraints in message sequence chart specifications. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X / PSTV XVII '97*, pages 91 – 106. Chapman & Hall, November 1997.
7. R.J.A. Buhr and C.S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.
8. Rational Software Corporation. UML notation guide. Research report, 1997. See also `http://www.rational.com/uml`.
9. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
10. Max Planck Institute for Computer Science. LEDA home-page, 1997. URL `http://www.mpi-sb.mpg.de/LEDA/leda.html`.
11. G. J. Holzman. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.
12. G. J. Holzmann. What's new in SPIN version 2.0. http://netlib.att.com/netlib/spin/index.html, 1996. Version April 17.
13. H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki. SDE: Incremental specification and development of communications software. *IEEE Transactions on Computers*, 40(4):553–561, Apr. 1991.
14. ITU-T. Recommendation Z.100: Specification and Description Language (SDL). Geneva, Switzerland, 1993.
15. ITU-T. Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1995. To appear.
16. ITU-T. Recommendation Z.120. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996. Review Draft Version.
17. I. Jacobson and et al. *Object-Oriented Software Engineering - A Use-case Driven Approach.* Addison-Wesley, 1992.
18. P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
19. S. Leue and P. B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proceedings of the 2nd Workshop on the SPIN Verification System, Rutgers University, August 5, 1996.* American Mathematical Society, DIMACS/32, 1997.
20. NTT Software Corporation, 223-I Yamashita-Cho Naka-Ku, Nakahama-Shi Kanagawa 23I Japan. *MuSiC++ Message Sequence Charts: How To Connect with SDL*, 1995.
21. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. L orensen. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.
22. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley & Sons, Inc., 1994.