Architecture of a Requirements and Design Tool Based on Message Sequence Charts

Hanêne Ben-Abdallah and **Stefan Leue**
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

Technical Report 96-13

hanene|sleue@swen.uwaterloo.ca

October 1996

Abstract

Message Sequence Charts (MSCs) are increasingly supported in software engineering tools to capture system requirements, test scenarios, and simulation traces of reactive systems. The latest standard syntax of MSCs [12] offers operations to compose MSCs in a hierarchical, iterating, and nondeterministic way. The various operators are a step towards increasing the applicability of MSCs to more than a trace language. However, current tools operate on MSCs that describe finite, deterministic behavior and none of them uses MSCs as a language for requirements specification and design of a system. In this paper, we propose an architecture for an MSC-based tool to support the requirements specification and design phases. The main functionalities of the tool are: an environment for the composition of system models through MSCs, syntactic and model-based analysis of an MSC model, and resolving resource related underspecifications in an MSC model. The proposed tool also supports synthesis mechanisms as a means to integrate our tool with available tools, e.g., the SPIN model-checker. In this paper, we also review the theoretical results we have currently developed towards realizing the proposed tool.

Contents

1 Introduction

The intuitive, graphical notation of Message Sequence Charts (MSCs) increased their popularity within the software engineering community. In particular, 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., [19],[10], [20], [13], [22], [6], [3], and [4]. MSCs are used to document system requirements that guide the system design [22], describe test scenarios (e.g., [13, 6]), express system properties that are verified against SDL specifications [3], visualize sample behavior of a simulated system specification [22, 3, 8], and to express legacy specifications in an intermediate representation that helps in software maintenance and reengineering [10].

In this paper we propose the architecture for an MSC-based tool for the requirements and design phases of reactive systems. The proposed 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 may be effective due to certain customizations. The integration is facilitated through the standardized syntax of MSCs by the ITU-T in Recommendation Z.120 [12].

A second motivation for the tool 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 [11], the MSC language offers modular and hierarchical operators for MSCs 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 it *environment* [23], communication exchanges, internal actions, and timers, and formal semantics [12, 14, 7]. 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 requires additional, explicit information, e.g., underlying network architecture and capacity and interprocess synchronization to resolve nondeterminism. For this, an MSC-based tool for the design of eractive 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 analysis, the tool must support deadlock detection, which is a common problem in communicating systems, and model-based analysis for properties such as safety and liveness. All of these analyses must be provided for MSC specifications with iterations and nondeterminism. Model-based analysis of MSC specifications is one area of tool integration where available, specialized tools, e.g., the SPIN model-checker [8] can be used without a need to develop new tools. Such a tool integration requires a translation from MSCs to the tool's language, e.g., Promela [9] for SPIN, and therefore must carefully address differences in semantics and expressiveness.

To summarize, our proposed tool extends the usage of MSCs in four ways: 1) provide an interface for modeling reactive systems through high-level MSCs; 2) facilitate the design of MSC specifications, e.g., by dealing with issues pertinent to the software and hardware architectures; 3) support static analysis for MSCs with iteration and branching; and 4) provide translation mechanisms to use available tools for model-based analyses as well as code synthesis. We have developed a prototype interface for modeling systems in terms of finite, deterministic MSCs. The theoretical grounds for static analysis and translation into Promela for model-based analysis through the SPIN toolset has been developed. We are currently augmenting the prototype interface according to the architecture we describe in this paper.

Paper organization. The subsequent Section briefly reviews the suitability of MSCs for requirements specification and design. Section 3 reviews current usages of MSCs in software engineering tools. Section 4 presents an architecture for an MSC-based tool for the requirements and design phases. Section 5 summarizes the paper and outlines future research directions.

2 Suitability of MSCs for Requirements and Design

The standard syntax of MSCs is defined by the ITU-T in Recommendation Z.120 [12]. An MSC 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 enforce timing constraints, create and terminate process instances. A consequence of using MSCs in industrial-size applications is the tendency to use them in a modular and hierarchical fashion just like other specification languages, e.g., RoomCharts [22]. For this, the standard Z.120 [11] evolved to allow the description of a large system by composing *basic* MSCs [12]. The resulting graphical language, called *High-Level MSCs* (hMSCs), 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.

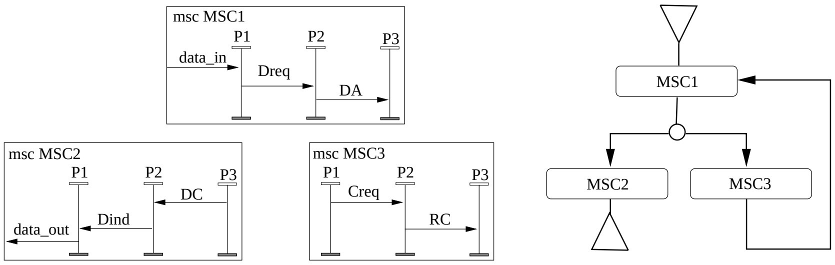

Figure 1: MSC specification example: basic MSCs (left) and high-level MSC (right)

As an example of 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.

Message Sequence Charts offer several advantages to the requirements and design modeling of reactive systems. One is the intuitive, graphical notation of MSCs which helps a designer to visualize the system's structure and interfaces. Another advantage of MSCs is the level of abstraction they offer by merely describing the message flow, 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 LOTOS, which explicitly specify the process behavior and leave the message flow implicit.

As a requirements language, MSCs offer the following concepts:

- Distinction between the *system* and its *environment* [23]. Reactive systems are often viewed as composed of the object software (or system) which must be developed and the environment where the system will be deployed and with which it interacts. MSCs offer the notion of an environment and provides the notion of *processes* to describe the system. For example, in Figure 1, the system is composed of three concurrent processes `P1`, `P2` and `P3`; the environment is represented by the rectangle enclosing the system.

- Distinction between the actions of the system and environment [23]. MSCs visually distinguishes between the actions the system produces or initiates from those produced by the environment. 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`. The visual distinction between the actions of the system and its environment facilitates the identification of the interface between the two components.

- Operational description of the system. MSCs describe the flow of actions in the system without assuming any implementation-related issues, e.g., processor allocation and speed, and network connectivity. In fact, an MSC specification can be interpreted as the set of admissible traces in a system [14]. It therefore can be used to validate or test a system specification, e.g., modeled in SDL [3].

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. However, to adopt fully MSCs for both the requirements and design phases, there are challenging issues that must be resolved and which stem from the implicit nature of process behavior in MSCs. More specifically, in the presence ofnon-determinism and iteration, explicit information is required about inter-process synchronization and the underlying network architecture and queuing strategies. The lack of this information may lead to discrepancies between an MSC specification and its interpretation and thus any potential implementation. In order to support iterative and nondeterministic composition of MSCs, it is therefore essential to detect the instances where underspecification is harmful, and to suggest to the designer possible extensions that resolve the discrepancies.

The strategies of current tools in addressing resource underspecification can be divided mainly into two classes. One class of tools only deals with finite, deterministic MSCs and thus avoids the problem by simply assuming an underlying network where all processes in an MSC can communicate with one another. A second class of tools provides a fixed, built-in network architecture and queuing strategy, mainly full-connectivity through FIFO queues with an infinite size. However, such an assumption is impractical and creates a gap between the requirements and design which, to some extent, guides the implementation. Only recently that an analyzer, POGA [4], was proposed to offer a designer more freedom in specifying the network architecture and queuing strategies. As we review shortly, this analyzer however operates on finite, deterministic MSCs that result from finite unfoldings of loops in the hMSC-like specification.

3 MSCs in Software Engineering Tools

We propose a set of requirements that we find important to hold of tools supporting MSCs as a main notation for requirements and design purposes.

1. The use of MSCs in the description of reactive systems makes constructs to support *branching* and *iterating* indispensable.

2. Although syntactic deviations can be tolerated, basic syntactic ideas defined in *Z.120* should be followed.

3. MSCs specify systems of asynchronously communicating concurrent processes, which is why MSC specifications represent *partial event orders* that should be interpreted as such in an MSC tool.

4. While a translation from MSCs into a different formalism for analysis purposes may be necessary, we require that there should be as little *semantic bias* as possible. In particular, we criticise the interpretation of MSCs based on SDL because of SDL's heavily constraining message passing semantics.

5. In particular in a notation that benefits from graphical appeal and visual allusion to such an extent as MSCs it is greatly necessary to have semantic assumptions explicitly represented in the specification. This applies especially to assumptions about the environment behavior and underlying communciation resources. Allowing non-explicit semantics assumptions would defeat the purpose of using MSCs. We call this the "*what-you-see-is-what-you-get*" (WYSIWYG) requirement wrt. the semantics given to MSCs in relation to their visual representation

6. A requirements tool needs to provide for means to check the *consistency* of the requirements specified. Syntactic property analysis is usually computationally less expensive than semantic analysis which is why we require that an MSC based design tool has at least semantic analysis capabilities, but ideally both.

7. Executing or *simulating* MSCs can greatly enhance the debugging of specification, which is why we suggest that a functionality should be provided by an MSC based tool.

ObjecTime

The ObjecTime tool (by ObjecTime Limited) uses basic MSCs in two ways: document requirements during the system description as a ROOM model; and visualize recorded traces of a ROOM model [22]. The behavior, i.e., execution steps of all actors within a ROOM model can be visualized as a basic MSC. The resulting basic MSC can be (manually) compared with the requirements MSC for a consistency check. An extension of the ObjecTime toolset has recently been suggested [21]. Given an arbitrary MSC, the processes of which correspond to actors in the ROOM model, it is possible to automatically synthesize the behavior of an actor that serves as a tester for the interaction of all other actors with the selected one.

Assessment. There is not treatment of infinite or branching MSC specifications on ObejcTime, and no automated support for verification or validation properties has so far been included (the test actor synthesis is not yet part of the tool).

GEODE

The AVALON toolset [3] within GEODE (by Verilog) uses MSCs in two ways: to visualize the dynamic behavior of an SDL specification, and to describe expected scenarios and tests that are verified against an SDL specification. The simulator in GEODE allows the user either to step through or exhaustively examine the behavior of an SDL specification. The tool supports a subset of SDL where processes have finite behavior [3]. Thus the produced MSCs are basic (finite), non-iterating MSCs. The second use of MSCs is to describe *observers* which witness the sequencing of events in an SDL specification. This is achieved in two modes: validation and verification. Validation ensures that the SDL specification exhibits only the sequence of events described by the MSC observer. Verification, on the other hand, tests whether the SDL specification has one execution as described by the MSC observer.

An MSC is interpreted according to a total ordering of its events. The total ordering is derived from: 1) the "local" ordering of events within each process; and 2) the "global" ordering of events as defined by their vertical distances from the start of the processes. The total ordering interpretation appears to be in contrast with the commonly accepted idea that MSCs describe partially ordered asynchronous systems. There are MSCs that are syntactically correct according to Z.120 but yet can not be interpreted in a meaningful fashion in GEODE (e.g., the sending of an event further down in the chart than its receiving). It is interestering to note that the SDL subset supported by the tool assumes process queues to be infinite [3]. This assumption frees the corresponding MSC description from issues related to queues.

Assessment. The presence of message queues violates our WYSIWYG requirement. Furthermore, we stated that MSCs should be interpreted as specifying partial event orders. GEODE enforces total order, which is counterintuitive and unnecessarily restricts the set of models that an MSC specification describes.

SDT

The SDL based tool SDT [1] knows two usages of MSCs. First, they can be used to visualize recorded finite execution traces of SDL specifications. Second, the tool allows for some limited requirements validation based on MSCs: it is possible to generate a validation model for the SDL specification, the validator will then check whether a given finite basic MSC represents at least one execution sequence of the validation model. The validation is thus based on a *trace inclusion* criterion: at least one of the traces in the high level requirement specification (expressed through a basic MSC) is required to be an admissible execution of the SDL system. The system is, however, allowed to have additional behaviors. Note that MSCs that have been generated by tracing an SDL specification can subsequently be used as requirement MSCs in the validation step.

Assessment. SDT provides for "testing" SDL specifications according to finite MSC specifications. The test, however, only checks for trace inclusion which allows for checking only a very limited number of properties. Liveness properties, for example, could not be checked with the SDT validator.

SDE and MuSiC++

These are two closely related tools. SDE [10] is based on a non-standard language called SAL that resembles MSCs. The tool provides three types of analysis: 1) *feasibility* analysis that performs a semantic reachability analysis to detect deadlocks, 2) *consistency* analysis that checks for 'non-resolvable' non-determinism (similar to our non-local choice branching analysis, see Section 4.2), and 3) *non-requested behavior*. Note that all analysis is done based on the generated SDL model which means that SDL's semantic assumptions (i.e., a process-unique input queue) are inherited into the MSC analysis. This may lead to the detection of deadlock situations that, under the assumption of multiple queues per process are no deadlocks.

In MuSiC++ [18] a synthesizer in the tool automatically generates executable SDL specifications from MSCs, that describe the "process view" of the system. The tool supports MSCs in accordance to Z.120 [11]. In terms of analysis, the tool supports MSC validation, consistency check between MSCs, and automatic test case generation. MSC validation is achieved by first generating the state space and then conducting reachability analysis to detect deadlocks, unreachable states, and unrequested behavior.

Assessment. In both tools, the dependency on the SDL semantics clearly limits the usefulness of the analysis, and the assumption of infinite queue sizes limits semantic verification. Furthermore, the assumption of the presence of queues violates our WYSIWYG requirement.

MSC Analyzer/POGA tool

This tool from Bell Labs [4] addresses problems of underspecification in MSCs specifications. It is based on the premise that the semantics of an MSC is influenced by implementation constraints, e.g., architecture and queuing policies. It provides MSC users with an analyzer that allows them to verify whether the visual ordering (i.e., intuitive semantics of an MSC) is enforced by particular architecture and queuing policies in an implementation. The semantics of a subset of basic MSCs is derived through three partial orderings: 1) the visual ordering of events within each process; 2) an "enforced" ordering which the implementation provides, possibly through forcing processes to wait; and 3) "inferred" ordering which the user might infer or assume based on the visual ordering. Both the enforced and inferred orderings are subsets of the visual order. The analyzer allows the user to explore a few built-in possible architectures and queuing policities and define their own. The analyzer automatically computes the enforced ordering and its transitive closure which is used to detect any *race condition*. A race condition is a discrepancy between the order of two events in the enforced and inferred orderings.

Assessment. POGA has been pioneering the tool supported analysis design based on MSCs. POGA also allows for some MSC composition, however, the composed MSCs remain finite traces through the hMSC graph, so that no complete reactive system specification can be given. The system lacks the possibility of checking general properties like liveness or consistency properties.

In summary, none of the tools reviewed above satisfactorily meets the requirements we outlined earlier.

4 Architecture of an MSC Requirements and Design Tool

The MSC-based tool consists of a GUI editor through which it offers four main functionalities: editing, syntactic analysis, model-based analysis, and code synthesis. Figure 2 presents a data flow diagram-like view of the tool we propose.

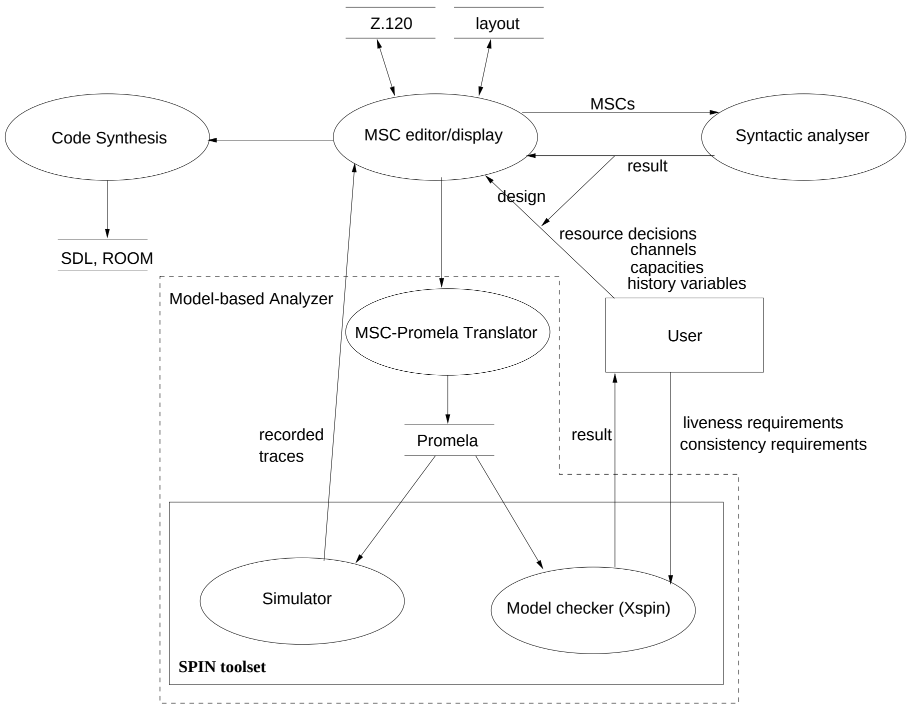

Figure 2: Architecture of proposed tool

4.1 Editing

The GUI interface allows the user to textually input, draw and manipulate MSC specifications. In addition, it allows the user to load and store MSC specifications. These editing functions provide syntax checking and graphics-support facilities that enforce the standard MSC syntax of Recommendation Z.120 [12].

Input/Output Formats. MSC specifications can be stored in different formats. In particular, out tool supports the following three options:

- *Z.120 textual format* [12]: This standard textual syntax facilitates specification sharing among tools. It claims to be equivalent to the graphical representation up to homomorphism in the sense that there is a `OUT` *message_type* - `IN` *message_type* statement pair with an identical message type in the textual representation whenever there is a signal arrow with that message type in the graphical representation.

- *Graphic Layout*: The textual representation of MSCs lacks layout information which is essential to reproduce chart layouts that the user has previously chosen. The graphic layout information essentially consists of the coordinates of an MSC's components.

- *Algebraic Format*: For various mathematical operations on MSCs, the second author has defined an algebraic representation for MSCs, called *Message Flow Graphs* (MFGs) [14]. The MSC-MFG translation is straightforward. It preserves the visual ordering of messages in the MSC specification. The algebraic notation is also useful in deriving an operational, finite-state semantics of an MSC specification, and thus facilitates the development of a visual simulator for MSCs.

Graphic Manipulation. The GUI provides an icon-based drawing palette with the basic components of the MSC language, e.g., processes, message arrows, labeling functions, and nodes in an hMSC. In addition, to facilitate modular description of an MSC specification, the GUI allows a user to select a node in hMSC and examine the corresponding MSC specification to which it refers. We have implemented a prototype editor for basic MSCs based on a tcl/tk GUI interface. We are currently augmenting the tool with graphics support for manipulating hMSCs.

4.2 Syntactic Property Analysis

It often less expensive to verify properties of a specification syntactically as opposed to analyzing the specification's model, a task that is inevitably exponential in the number of processes and communication events in the system. In addition to syntactic well-formedness of an MSC specification, MSC specifications can be syntactically analyzed for: deadlocks, race conditions, process divergence, and non-local branching choices.

Deadlocks. Based on a result by Ladkin and Simons [15], deadlock detection for bMSCs reduces to the detection of cycles in message flow graphs. For an example of a bMSC with a deadlock see Figure 3 (a). The algorithm for cycle detection (modified Tarjan's depth first search [2]) is linear in the number of nodes in the graph. For a complete MSC specification it is easy to show that it is deadlock-free in case all of its bMSCs are deadlock-free.

Race conditions. A race condition in MSCs describes the occurrence of two events in an order that is different from their visual ordering. This phenomenon results from the underspecification of hardware architecture and queuing strategies in MSCs. As we reviewed in Section 3, the POGA analyzer [4] can syntactically determine whether a basic MSC has a race condition. The basic MSC can be the result of user-selected sequence of bMSCs and finite loop-unfoldings in an hMSC. Once the POGA software is accessible, we can easily integrate it into our tool.

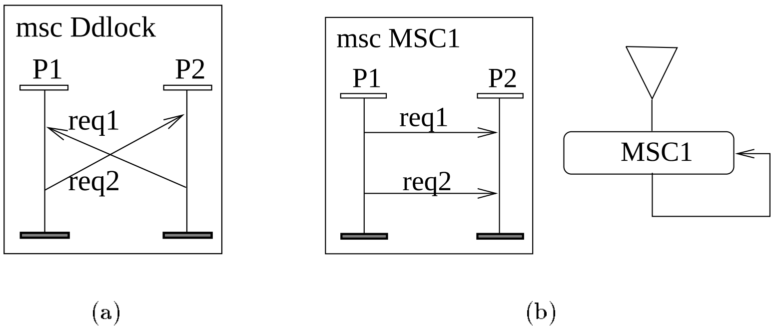

Figure 3: (a) Deadlocked bMSC; (b) MSC specification with a process divergence

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 3 (b). One possible execution of this MSC specification is the infinite trace `!req1 !req2 !req1 !req2` ··· 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? How are multiple copies of a not-yet received message handled?

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. Furthermore, while the above questions seem pertinent to the implementation phase, we view process divergence as *unintended* behavior of the specification that must be rather 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 which refines the specification.

Process divergence can lead to discrepancies between the specification and implementation, e.g., message over-writing and unexpected deadlocks, as well as unimplementable specifications, e.g., one that requires message queues with infinite sizes. It is therefore essential to detect potential process divergences in an MSC specification prior to implementation.

We have syntactically characterized process divergence and developed an algorithm 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 (thus constitute one thread of execution) and verifies that the processes within the bMSCs communicate through a hand-shake.

Non-local branching. An MSC specification can compose basic MSCs to express alternative behavior. 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. 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. In terms of interpretation, this assumption may result in an infinite state space.

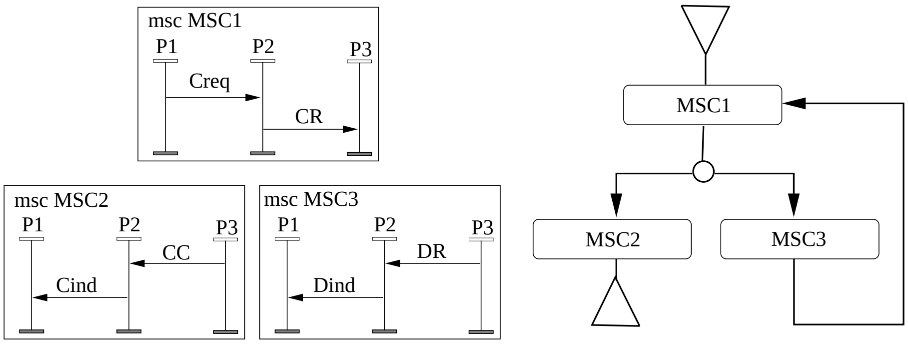

Figure 4: MSC Specification with a local branching choice

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 MSC1. 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 [12] nor hMSC graphs provide an explicit way to handle such an information exchange. To handle this type of inter-process synchronization, Ladkin and Leue [16] suggested the use of global history variables that keep track of early process branching choices. Their approach, however, can result in an infinite-state semantic representation (i.e., global system transition graph) which, on one hand, is counter-intuitive to the visual interpretation of an MSC specification and, on the other hand, can impede formal analysis.

However, not all branchings in an MSC specification require global history variables to keep track of early process branching choices. Consider for instance the MSC specification in Figure 4. In this example, the type of the first received message can be used to determine the choices made by other processes in the specification. Consider process P3; since sending messages is non-blocking, this process can decide to proceed either as MSC2 or MSC3 independently of other processes. It can therefore either send message CC or DR, respectively, by making a local decision to resolve the non-determinism. On the other hand, since the first event in process P2 is to receive either message from P3, process P2 can learn about the decision that P3 made based on the type of message it receives: if it receives a CC message, it knows that the MSC2-branch has been chosen and proceeds with sending a Cind to P1; otherwise it receives a DR message, knows that a branching to MSC3 has occurred, and follows accordingly by sending a Dind to P1. Finally, process P1 can also resolve the non-determinism based on the type of message it receives from process P2. This strategy of *wait-and-see* can be easily implemented and eliminates the need for global history variables.

We have syntacticly characterized non-local branching choice and developed an algorithm that execute 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.

Analysis Results. Results of the syntactic analysis will be displayed in the graphical image of the MSC specification by 'tagging' the parts of an MSC that violate a condition. For deadlocks, the cycles detected in a bMSC are highlighted; for a race conditions, the involved messages are highlighted; for process divergence, the cycle in the hMSC along which divergence occurs as well as the sets of processes that diverge are highlighted; and for a non-local choice, the bMSCs where a non-local choice occurs is highlighted.

After tagging a problem in the graphical image, the user ca edit the problematic components. The editor can in some situations make suggestions for resolving a problem, e.g., by introducing additional message arrows to avoid process divergence [5].

Adding Resources. Process divergence and non-local choice situations in MSC specifications can be harmful: at best they indicate the underspecification of important detail, at worst they lead to models that have no meaningful interpretation, e.g., deadlocks resulting from non-local choices. The syntactic analysis should therefore be followed by a modification, or refinement, of the MSC specification to resolve such unintended behavior. As we suggest in [5], the resolution can be through adding messages. However, this may not always be possible in which case the user needs to provide additional information about the system resources. For instance, to deal with potential race conditions, process divergence and non-local choices, the user can specify the following two resources:

- *Channels*: The user indicates the presence of queues that store messages sent but not yet received. The user can specify one channel per message arrow, or group a set of message arrows to be served by the same channel. Along with the location of a channel, the user specifies the capacity and queuing protocol of the channel.

- *History variables*: For each branching in the hMSC, the user indicates the presence of history a variable with its capacity. History variables are used to encode the decisions made by processes to resolve non-determinisms.

Figure 5 shows the suggested additions as in [16]. Channels are explicitly specified by overlaying message arrows with a suitable symbol; channel capacities are indicated under eacxh channel and when missing they are assumed to be infinite; and a history variable (an instance of which will exist for each pair of directly communicating processes) with a finite capacity has been indicated inside the branching point in the hMSC. In this example, each message has its own dedicated channel and each channel is assumed to be a FIFO queue.

The above resource requirements are accounted for when interpreting an MSC specification. They refine the requirements by making them more realistic and closer to a potential implementation. After specifying the resource requirements, the user can re-analyze the design to verify that the resource requirements are satisfactory, e.g., do not produce a race condition, or buffer overflow.

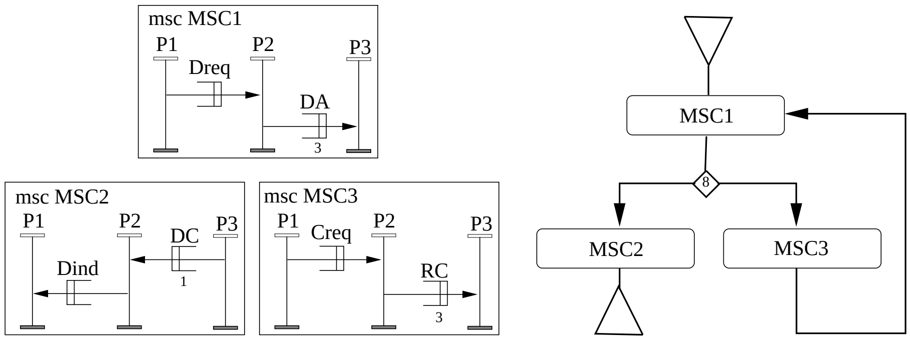

Figure 5: MSC specification with explicit channels, channel capacities, and finite length history variable

The current MSC standard Z.120 does not support explicit description of resources. The suggested notation in [16] and as illustrated in the above example is an easy and unambiguous extension to the standard Z.120.

4.3 Model-Based Analysis

A model-based analysis of an MSC specification is an analysis over its interpretation, i.e., state space model. While a model-based analysis is often more expensive than a syntactic analysis, it allows the verification of more functional properties of the system. Two particularly useful model-based analyses are simulation and model-checking of logic properties. To support these two types of model-based analyses in our tool, we will make use of the SPIN toolset [8] along with an existing synthesis technique [16] between MSCs and Promela [9], the modeling formalism in SPIN.

Informally, an MSC specification is translated into Promela as follows: processes in the MSC specification are mapped into concurrent Promela processes; each message arrow is mapped into a communication channel with a capacity $= 1$; and each communication event in the MSC specification is translated into a Promela communication statement with a special care to ensure the event atomicity. Branchings and iterations in the MSC specification are modeled through labels and goto statements in Promela. The MSC to Promela translation in [16] illustrates how to introduce channels with capacities greater than 1. It also describes how to implement a history variable-based synchronization algorithm to execute an MSC specification with non-local choices.

4.3.1 Simulation

Simulating a specification is vital for debugging and better understanding specifications. At an early stage of our tool development, we will take advantage of the simulation capabilities of SPIN and obtain our simulation from a translation into Promela followed by an invocation of SPIN. At later stages we plan to develop an independent simulator.

Figure 6 shows an execution trace that was obtained from the MSC specification in Figure 4 using the MSC to Promela translation algorithm described in [16]. The left-most vertical line represents the SPIN run-time system, and the remaining ones correspond to the MSC processes. Each box represents an MSC communication event, and each arrow indicates a message exchange. The shown trace illustrates an initial handshake where process P3 refuses the connection establishment through a DR message, and then a handshake that leads to a successful connection establishment.

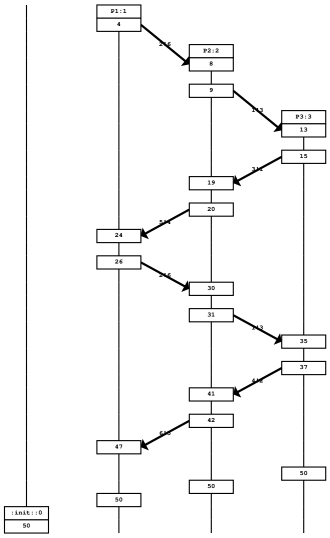

Figure 6: MSC execution trace generated from MSC specification

Note that it is fruitful to allow for feeding the traces generated by the simulation component as bMSCs back into the MSC specification maintained by our tool. This way, new scenarios that might represent particularly interesting executions of the original MSC specification can be added to its set of bMSCs.

4.3.2 Semantic Property Analysis

The semantic analysis of properties complements the syntactic analysis with more functional properties that include:

Liveness properties: These require that eventually something good will happen. For instance, in the example of Figure 4 we may require that if P1 has sent a Creq message (written as $!Creq$), then *eventually* either a message Cind ($?Cind$) or a message Dind ($?Dind$) will be received. Using Manna-Pnueli linear time temporal logic (LTL) [17], this requirement can be formalised as $\Box !Creq \supset \Diamond(?Dind \vee ?Cind)$. Note that this is liveness property on the service provided by P1 is *not* expressed by the MSC specification. It therefore complements the MSC specification.

Consistency properties: These state that certain events do not happen along an execution trace in a particular order. In the example of Figure 4 we may require that once the communication partner P3 has declined to establish a connection ($!DR$) it will never be possible that the process P2 signals a connection establishment ($!Cind$) to the user process P1. This can be formalised as $\Box !DR \supset \neg\Diamond !Dind$ and is clearly satisfied by the Example in Figure 4.

We will make use of the MSC to Promela translation mechanism to benefit from the efficiency of the model-checker in SPIN. This latter allows the user to enter an LTL property that is validated against the Promela specification. If the Promela (and thus MSC) specification does not satisfy the LTL property, SPIN produces a trace to illustrate the mismatch. The above two sample properties have been validated through SPIN.

An important issue in model-checking as well as any model-based analysis is dealing with the size of the state space. To obtain a finite-state MSC specification, the user may need to augment the MSC specification with resource information as described in the previous section.

4.4 Code Synthesis

In addition to the translation to Promela which we described earlier, we will examine translations into other languages in order to produce executable code, e.g., SDL and C++ code. ObjecTime has done preliminary work on translating bMSCs into ROOM actor code [21]. This translation however requires further work since it focuses on the synthesis of test actors, i.e., finite and non-branching MSCs.

An important issue in any automated code synthesis is the quality of the produced code. MSCs are aluable design descriptions when it comes to focus on the communication behavior of a process. Thus, any reasonable synthesis will produce accurate code that completely describe the communication in the system. However, since process behavior is implicit within MSCs, any automated synthesis can at best produce a skeleton code for the processes. The internal actions of a process will therefore have to be manually added to the synthesized code.

5 Conclusion

We have proposed an architecture of a tool for the requirements specification and design of reactive systems. The underlying formalism of the tool is Message Sequence Charts, a graphical language that is gaining popularity within software engineering methodologies and tools. The proposed tool architecture extends the usage of MSCs as a requirements and design language through three main functionalities: graphical editing, analysis, and synthesis. We have reviewed the theory underlying the analysis functionality, as well as a way to incorporate resource requirements into an MSC specification in a step towards deriving a design.

We have designed a prototype GUI that supports the drawing, saving and loading of basic MSCs. We are currently augmenting the prototype with a graphics support for high-level MSCs, the syntactic analysis described in this paper, as well as an automated synthesis mechanism to translate into Promela and make use of the SPIN model-checker. Other future research issues are the interpretation of MSCs within a timed model and the support of notions of refinement and abstraction to incorporate top-down or bottom-up design with MSCs.

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.

[2] A. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and ANalysis of Computer Algorithms*, chapter 5. Addison-Wesly Publishing Company, 1974.

[3] 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.

[4] 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.

[5] H. Ben-Abdallah and S. Leue. Syntactic analysis of Message Sequence Chart specifications. Tech Report 96-12, Department of Electrical and Computer Engineering, University of Waterloo, 1996.

[6] R.J.A. Buhr and C.S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.

[7] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definitions for message sequence charts. In O. Faergemand and A. Sarma, editors, *Proceedings of the 6th SDL Forum, SDL'93: Using Objects*, October 1993.

[8] G. J. Holzmann. What's new in SPIN version 2.0. http://netlib.att.com/netlib/spin/index.html. Version April 17, 1996.

[9] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[10] 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.

[11] ITU-T. Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1995.

[12] ITU-T. Recommendation Z.120. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996. Review Draft Version.

[13] I. Jacobson and et al. *Object-Oriented Software Engineering - A Use-case Driven Approach.* Addison-Wesley, 1992.

[14] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[15] P. B. Ladkin and B. B. Simons. Static analysis of communicating processes. To appear, Springer Lecture Notes in Computer Science.

[16] S. Leue and P. B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. In J.-C. Grégoire, G. Holzmann, and D. Peled, editors, *Proceedings of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System, Rutgers University, August 1996*. American Mathematical Society, 1997, to appear.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[18] NTT Software Corporation, 223-I Yamashita-Cho Naka-Ku, Nakahama-Shi Kanagawa 23I Japan. *MuSiC++ Message Sequence Charts: How To Connect with SDL*, 1995.

[19] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92*. Elsevier Science B.V. (North-Holland), 1994.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.

[21] B. Selic. Automatic generation of test drivers from MSC specs. Technical Report TR 960514 - Rev. 01, ObjecTime Limited, Kanata, Ontario, Canada, 1996.

[22] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons, Inc., 1994.

[23] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, January 1997. (to appear).