FI MU

Faculty of Informatics
Masaryk University Brno

Formalisms and Tools for Design and Specification of Network Protocols

by

Jindřich Babica
Vojtěch Řehák
Petr Slovák
Pavel Troubil
Martin Zavadil

FI MU Report Series FIMU-RS-2007-02

May 2007

Formalisms and Tools for Design and Specification of Network Protocols[*]

Jindřich Babica Vojtěch Řehák Petr Slovák
Pavel Troubil Martin Zavadil

Faculty of Informatics, Masaryk University

Botanická 68a, 602 00 Brno, Czech Republic

{xbabica,rehak,xslovak2,xtroubil,xzavadi2}@fi.muni.cz

May 30, 2007

Abstract

Message Sequence Charts (MSC) are a useful formalism for formalization of network protocols early in their design phase. In this paper, we introduce the basics of MSC language and describe some of the possibilities for automatic location of "problematic" parts in the design. Focus is then given to different modifications of MSC design (FIFO behavior, bounded channels, etc.) as well as formal checking of more complex design properties (MSC membership, realizability). Next, an introduction of Specification and Description Language (SDL) is presented. Possibilities of automatic synthesis of system design in MSC to an SDL model and it's correctness verification are mentioned.

1 Introduction

Recent advances in technology make it possible to design very sophisticated and complex systems (hardware systems, software systems or their combination). As experience

[*]The work presented in this report has been supported by the research centre "Institute for Thoeretical Computer Science" (ITI), project No. 1M0545, and is a result of cooperation between Faculty of Informatics, Masaryk University, and ANF DATA spol. s r.o.

shows, the main problem of such systems is the difficulty to achieve a good system design. This problem is accentuated especially in complex systems, where, as the complexity of a system grows, so does the probability of flaws in its design. Each of these design errors, if not detected early enough, can lead to a significant expense increase during the implementation stage.

Nevertheless, software companies usually carry out most of their verification and testing processes during the end phases of the product's implementation stage. The reason for this (possibly expensive) behavior is frequently the absence of a formal design specification. It is not unusual for the design process to be carried out in natural language only. Although it is possible to have such natural language specification appropriately structured as well as different levels of granularity provided, it cannot serve as a basis for (semi)automated formal verification.

A development of design specification in a formal language is often seen as an unnecessary work, especially if no design error is found. Even in these situations, formal design specification has distinctive benefits. Precise and unambiguous semantics of formal languages make it possible to have unambiguous specification of the sought system – natural languages with their inherent ambiguity can't guarantee this feature at all. Also, machine-readable design can be used for automated verification as well as other implementation help. These include anything from automated creation of header files to automatic generation of the whole system (system synthesis).

The structure of this paper is as follows: First, MSC language is introduced and possibilities of basic syntax properties checking are mentioned. With this knowledge, more complex properties as MSC membership and realizability are discussed in Section 4. Focus is then given to the possibilities of automatic conversion (synthesis) of a system design in MSC to other formal languages. Specification and Description Language (SDL) is introduced in Section 7. The paper concludes with a description of possibilities for synthesis from MSC to SDL.

2 Message Sequence Charts

Message Sequence Chart (MSC) is a graphical and textual language for the description and specification of the interactions between system components. Message Sequence Charts are mainly used as a specification of real-time system behavior, in particular of telecommunication switching systems. Message Sequence Charts may be used for requirement, interface and test-case specification, simulation, validation and documentation of real-time systems.

The main idea of MSC is to model the system by a set of individual **system runs** consisting of message exchanges between processes. A **run** is a explicitly specified determinist sequence of communication acts between processes. No knowledge of inner workings of processes is needed or expected.

Core MSC language is called Basic Message Sequence Chart. A Basic Message Sequence Chart (bMSC) describes a sequence of communication exchanges between a set of processes in the system. Advanced descriptive possibilities are provided by High-level Message Sequence Charts (HMSC). In contrast to bMSC, High-level MSC is intended as a description of relations between bMSCs in the sense of particular sequence of execution of individual bMSC. Combining bMSC and HMSC, system designer provides set of possible runs of system.

Note again that MSC contains both textual and graphical form and their expressiveness is equivalent. To ease the readability of this document, we have decided to use just the graphical form. For textual form syntax or other details please follow [15].

2.1 Basic Message Sequence Charts

Basic MSC is formed by a finite collection of process instances (or processes) surrounded by frame with name of MSC (see Figure 1). An instance can be drawn in two ways shown in Figure 2. The reason for this duality will be explained later.

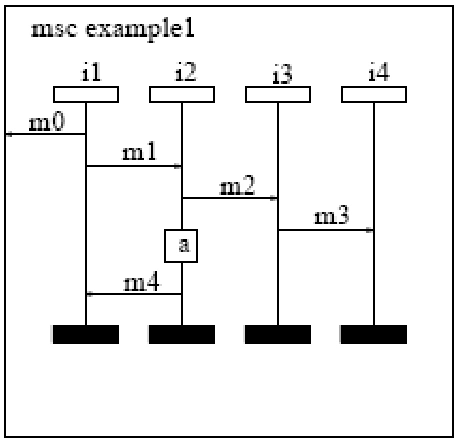

Figure 1: Basic MSC example

Message transmissions (with a particular name) are depicted as labeled arrows. They start at sending instance (send or outgoing event) and end at receiving instance

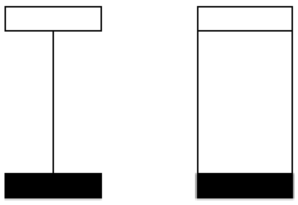

Figure 2: Two ways of drawing instances

(receive or incoming event). Local action is denoted by rectangle with action string inside. Action string has no special semantic.

Time progress and therefore also order of events on one instance is from top to down. It is assumed that all events (message output, message input, local action) consume no time but delay between two succeeding events can be completely arbitrary. No global notion of time in the system is assumed.

It is possible to describe process creation and termination by Basic MSC (shown in Figure 3). Dashed arrow is called create-line symbol.

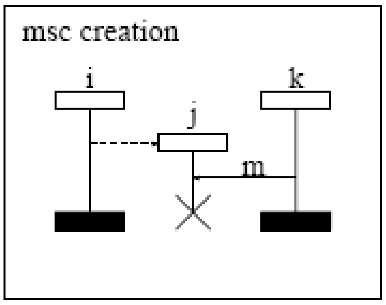

Figure 3: Creation of process

Timer handling is also very simple to describe, see Figure 4. The first couple in this picture shows set event of timer-horizontal or bent line to hourglass with name of timer. Parameters can be passed to timer in set event but have no semantic meaning. Second picture in Figure 4 depicts reset of timer and the last one shows timeout.

It is possible in Basic MSC to define a message to be lost or spontaneously found – see Figure 5 for syntax details.

Conditions are also supported by MSC. Generally, these conditions have no specific semantic meaning. They can be used as a label for a specific situation that has occurred on the instance and therefore improve bMSC's readability. Example of condition which holds for more than one instance is shown in Figure 6. It describes a situation, when condition C holds for instances i, k but not for j (this is denoted by j's axis drawn through).

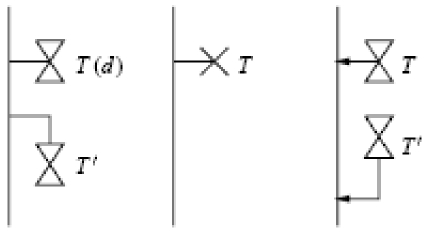

Figure 4: Timer handling example

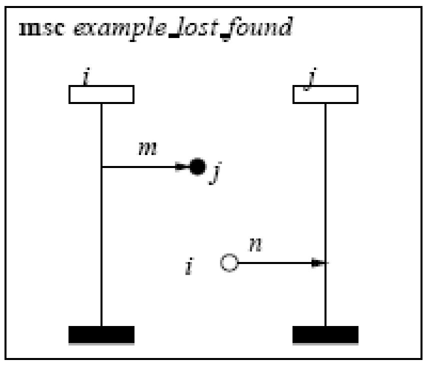

Figure 5: Lost/found message

2.1.1 Ordering facilities

So far, events on an instance were totally ordered in time. Sometimes it might be useful to be able to specify, that some events on the same instance may be completely unordered. This can be defined in MSC as a coregion. An example of a possible co-region usage is in Figure 7. In this figure incoming event of message m and outgoing event of message n are unordered on instance i but they are executed after output of message k and input of message l.

Because input of a message arises naturally after its output, we can use this information to get a partial order of all events across different instances. Of course, because

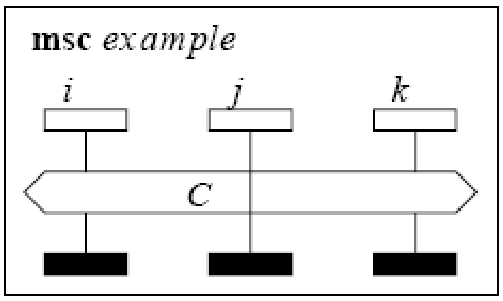

Figure 6: Condition example

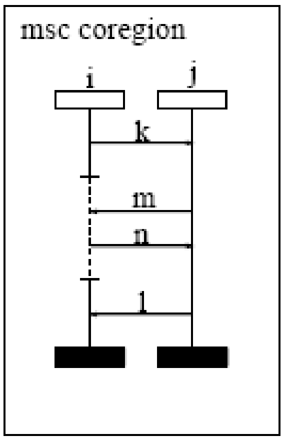

Figure 7: Co-region example

not all processes are bind together by message exchange, some events may be still unordered. Above this partial order, other order relations emerging for example from unmodeled parts of system might be explicitly specified by an MSC construct called general ordering. An example can be seen in Figure 8 where the local action a at instance k occurs after output of message m (not necessarily immediately after the output). Figure 9 depicts use of general ordering in a co-region. It shows that input event of message m and output event of n and o are generally unordered except that o must occur after m.

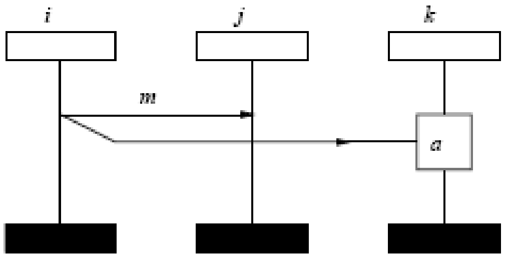

Figure 8: General ordering example

2.1.2 Vertical, horizontal and alternative composition

MSC specification provides several operators for composing MSCs (HMSC as well as bMSC). These operators are vertical composition (operator seq), horizontal composition (operator par) and alternative composition (operator alt).

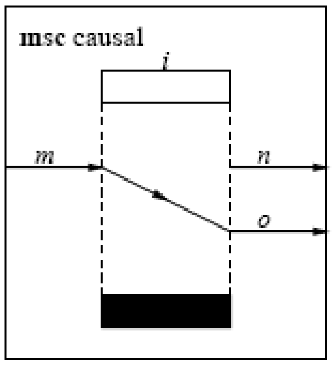

Figure 9: General ordering in co-region

Composing two MSCs using operator seq results in an MSC where all events from an instance of the second MSC have to occur after the events from the same instance of the first MSC.

Horizontal composition of two MSCs provides MSC which exhibits interleaving behavior of first MSC and the second one on common instances. Example is shown in Figure 10.

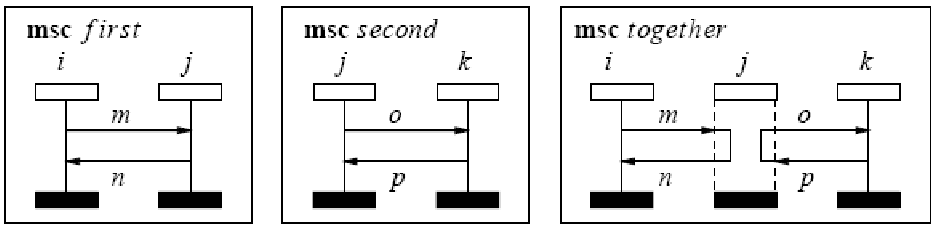

Figure 10: Horizontal composition

Alternative composition is used to describe several possible behavior of system. For example, if A and B are MSCs, an expression A alt B means that A xor B is executed.

2.1.3 Inline expressions

Inline expressions are a way of describing nontrivial sets of system runs while keeping the MSC diagrams simple. To depict alternative or horizontal composition we use inline expressions in a way shown in Figure 11. The first MSC indicates that output and input of message m occurs on instances i, j or output and input of message n occurs on the same ones. The second one indicates that the two parts delimited by dashed line are interleaved when executed.

The MSC language gives us also the opportunity to describe a situation when a part of MSC is executed a specified number of times (e.g., anything from 2 to infinity, 3 to 5 times, etc.). This can be also written as an inline expression, more precisely as a loop expression which is followed by loop boundary. Loop boundary refers to the number of possible repeated vertical composition of content of this inline expression and indicates minimal and maximal number of repetitions. Figure 12 shows usage of loop inline expression and its semantically equivalent MSC.

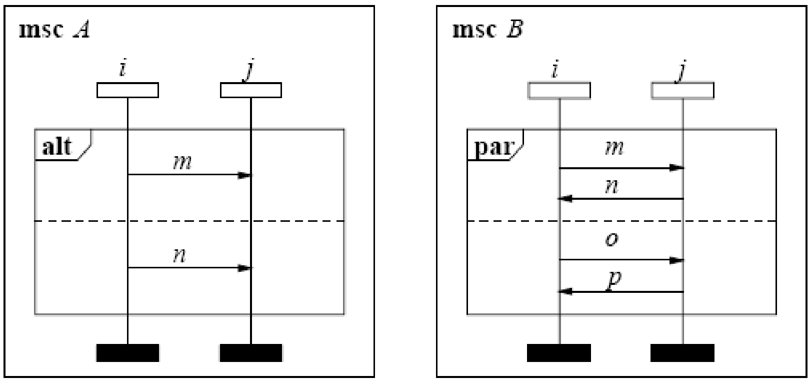

Figure 11: Inline expressions

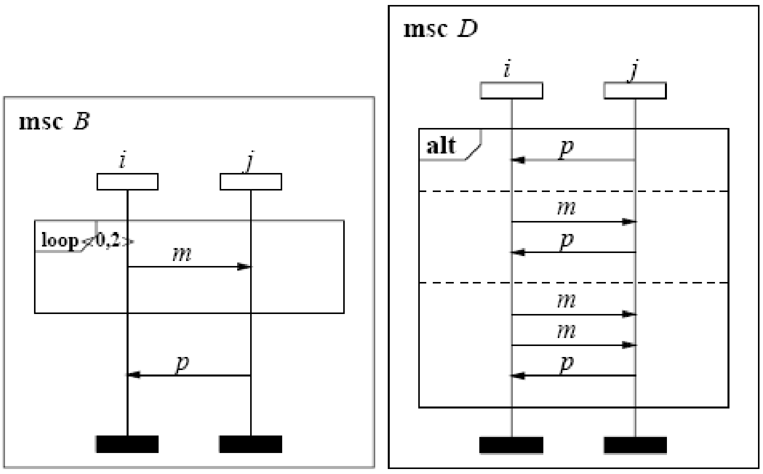

Figure 12: Loop inline expression

2.1.4 MSC reference expressions

Real systems are often very complex and it is therefore natural requirement for a specification language to offer tools to cope this complexity. MSC reference expressions are one of these tools.

MSC reference expressions are used to refer to an another MSC by using it's name. The basic idea is that referencing an MSC is semantically equivalent to pasting the whole MSC to the same place. It is allowed to use composition operator (alt, seq, par) inside reference expression. Figure 13 depicts example of MSC reference expressions' usage.

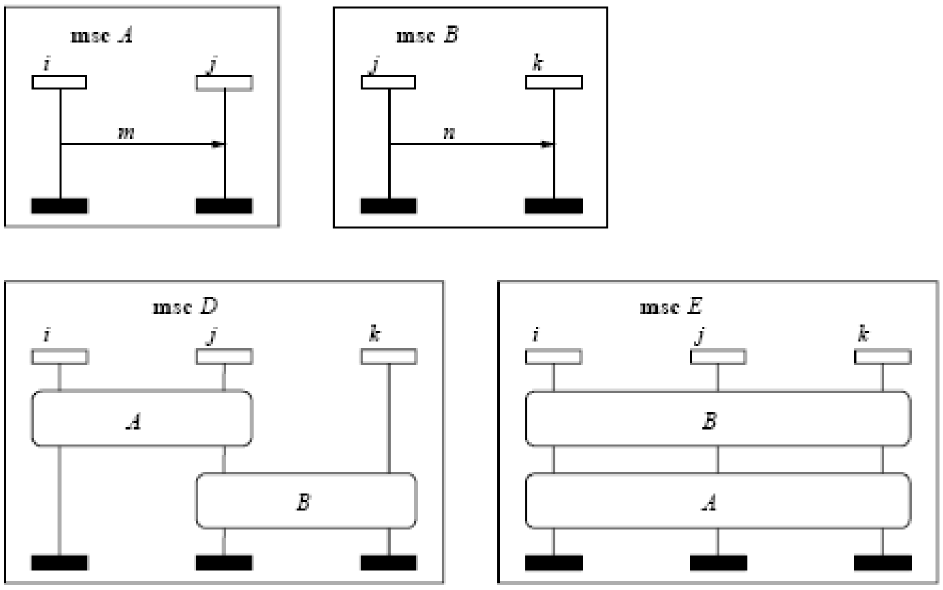

Figure 13: MSC reference expressions

2.1.5 Gates

MSC recommendation describes how to define an interface of individual bMSC. For this purpose gates are used. There are two types of gates: message gates and order gates. Message gates are used for message events and order gates are used for ordering of events.

Input and output gates are depicted as message arrows (order arrows) connected to surrounding frame of bMSC where the name of gate is presented. Example of gates' usage is shown in Figure 14. Note that it is allowed to send message x to gate y (and conversely) of reference expression if and only if there is an input gate y for message x in the referred MSC.

2.2 High-level MSC

High-level MSC provides an easy and transparent way how to combine several MSC together. Syntactically, High-level MSC is a directed graph where the nodes represent

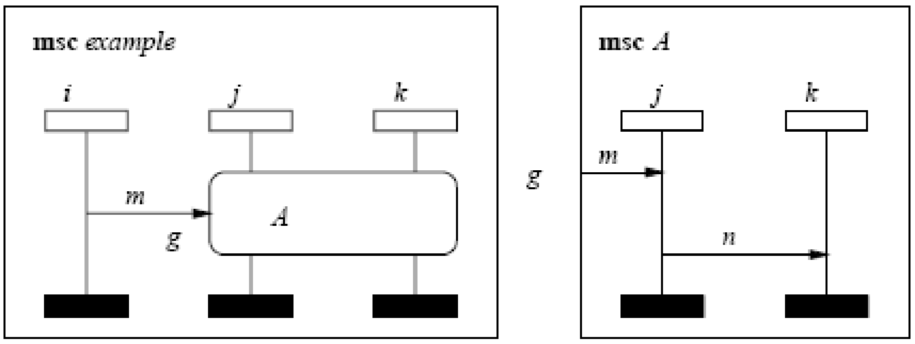

Figure 14: Gates' usage

other MSCs and vertices imply an order of the nodes. Nesting of MSCs is allowed only in its finite form (recursive nesting is forbidden). One MSC can be included by more than one node of HMSC – this makes it possible for individual MSCs to be called similarly as functions in programming languages. An HMSC provides also other elements:

- start node ▽
- end node △
- msc reference node ▭
- condition node ⬡
- connection node ○
- parallel frame ▭

Start node, mandatory in every HMSC, determines initial point of HMSC. Analogically, end node indicates the terminal node of HMSC. MSC reference node refers to other HMSC or bMSC by its name. See Figure 15 for example of HMSC notation.

Semantic of HMSC is easy to explain by operators introduced so far and recursive substitution of graph vertices with corresponding MSCs. If nodes are connected via one arrow (edge) they are exactly vertically composed. If a node has more than one outgoing arrow, then all following nodes are alternatives for the vertical composition with ancestor node. Parallel frame indicates horizontal composition of its content. Condition node has no defined semantic and connection node disambiguates crossing lines from splitting lines.

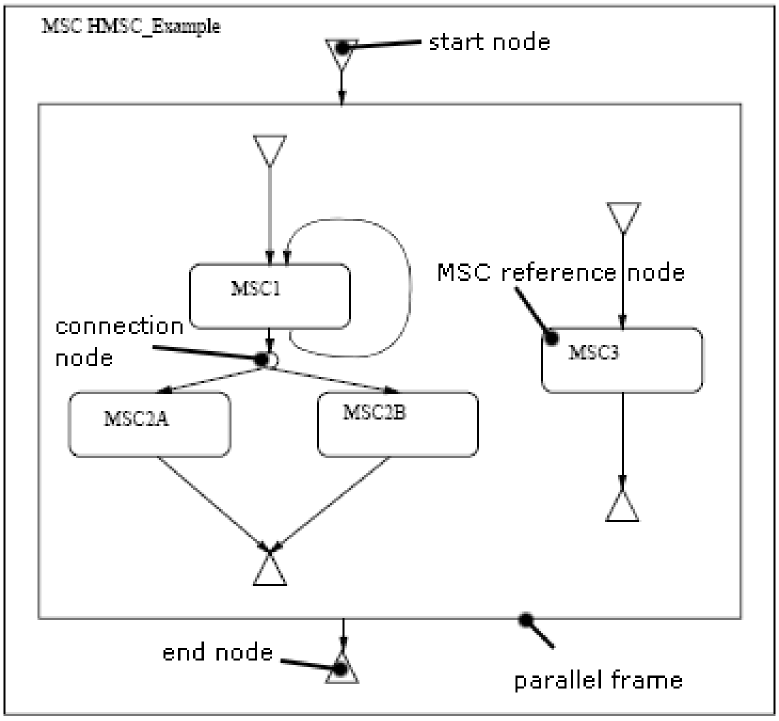

Figure 15: HMSC example

2.3 Overview

We have shortly introduced general facilities of MSC and informally described their semantic. For formal semantic please see ITU-T Recommendation Z.120 (Annex B) by which this text was inspired.

MSC provides wide spectrum of features for describe system behavior. Much less positive is tools' support of these features. There are only a few tools (commercial mostly) that provide almost all features described in MSC formal specification.

Single bMSC or sequence of bMSCs (described by HMSC) should represent intended system behavior (its runs). MSC assumes as little as possible about inner structure of individual processes and the environment surrounding the system. It only states, that each event at each instance should be performed by target component in the same order as is specified by MSC.

System behavior described by MSC may introduce some inconsistency to implementation of the system (in many possible ways), eventually the proposed system needn't to be even realizable due to these inconsistencies. Therefore it is necessary to check MSC's design properties preferably through the use of automatic checking methods.

3 Checking Basic Syntactic Features of MSC

In this section, we focus on some MSC syntax features that point out suspicious parts of the design where a design error could have occurred. With the knowledge of the system behavior we can analyze other MSC properties described in section Section 4.

One of the tools designed to check these syntax features is an MSC analyzer MSCan [23]. It can analyze a textual representation of MSC given on input for various properties. The main properties are as follows.

3.1 FIFO

FIFO property ensures that there is no overtaking in message channels. FIFO is the most important feature for analyzing other MSC properties (such as realizability). An example of MSC that hasn't got FIFO feature is shown in Figure 16.

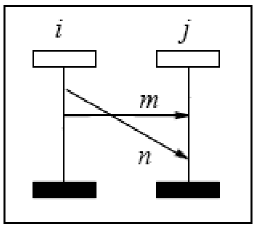

Figure 16: Non-FIFO

3.2 Acyclic

Acyclic property ensures that there is no cycle in the ordering relation on events for every MSC (event = send or receive a message). If there is a cycle then there is a message which is received earlier than it is sent. Cyclic behavior is not correct and such system is not implementable. An example of cyclic behavior is shown in Figure 17.

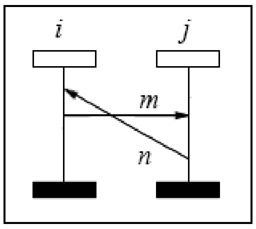

Figure 17: Non-acyclic

3.3 Safety

A MSC is called safe, if every sequence of nodes describing an accepting path in graph results in correct MSC. In other words, every sent and lost message is found and received later. Non-safety feature leads to incorrect MSC and the system behavior is not implementable. An example of MSC that hasn't got safety feature is shown in Figure 18.

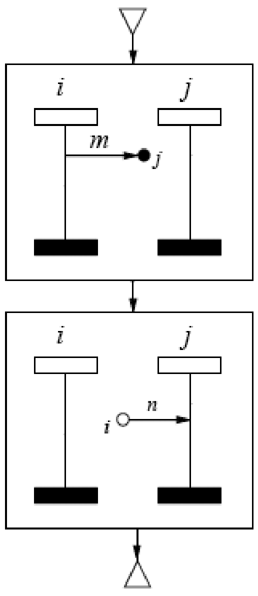

Figure 18: Non-safe

3.4 Local Choice

Local Choice property ensures that the communication is initiated by exactly one process for every node after branching. The process is active in foregoing node moreover. This feature can be verified by checking if there is exactly one minimal event in the ordering relation. Non-local-choice in MSC specification may lead in implementation to deadlock (see Subsection 4.5). An example of MSC that hasn't got local-choice feature is shown in Figure 19.

3.5 Local Cooperativity

Intuitively, an HMSC is not Local Cooperative if it contains at least two sequentially connected MSCs, that are describing communication on disjoint system parts. Figure 20 depicts a non-local cooperative HMSC.

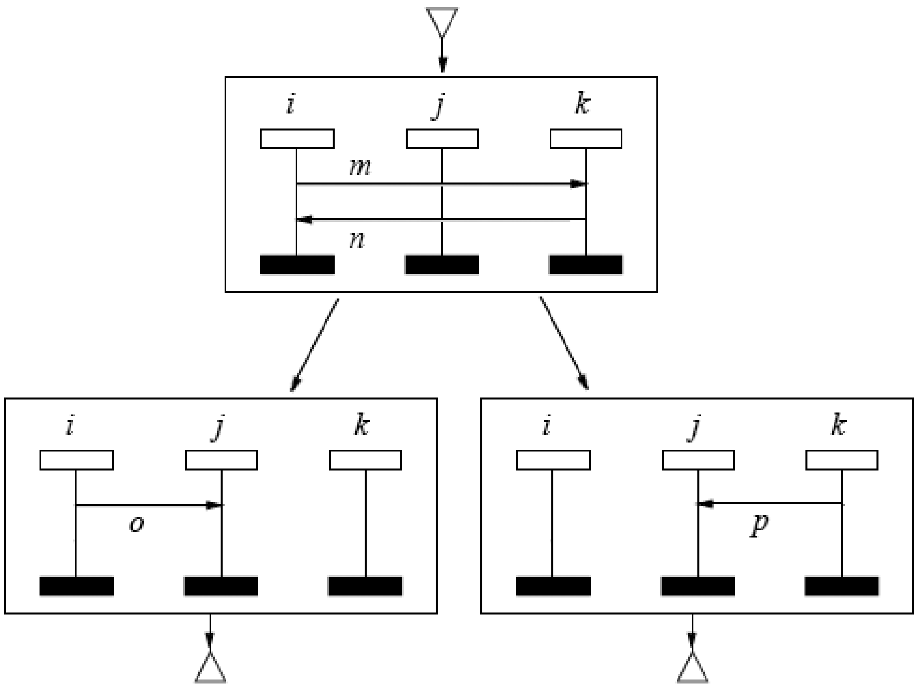

Figure 19: Non-local-choice

3.6 Boundedness (regularity)

Intuitively, if an HMSC is bounded, only finite buffers are needed in the system. Said in another way, only finite number of messages can be send before a reply is received.

Formally, an HMSC G is **bounded** (or regular) iff for every cycle in G holds: Denote H a graph with communicating processes in the cycle as nodes and oriented edges representing communication in this cycle between the processes. Then H is strongly connected.

Examples of HMSCs that are (un)bounded are shown in Figure 21 and Figure 22. This property is tightly connected to possible verification of system design (see Subsection 4.3 for more details).

3.7 Race Conditions

System behavior as described by MSC can differ from implemented system behavior that is embedded into certain environment. Target environment may influence many factors that system behavior depends on. One of these factors is ordering of message delivering.

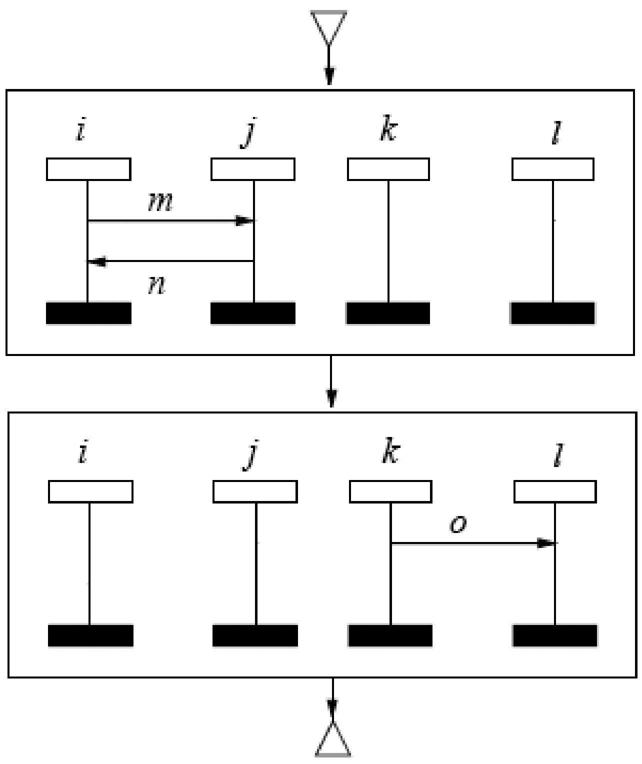

Figure 20: Non-locally cooperative HMSC

The paper [3] introduces an approach for detecting locations in bMSC (it's restricted form) where different ordering of events in bMSC and real system can arise. For this purpose they introduce four semantics of environment which messages are sent through.

- **Single FIFO queue per process**: Each process has single FIFO queue for all received messages.
- **One FIFO queue per source**: Each process p has a single FIFO queue for each process which p can receive message from.
- **Single non-FIFO queue per process**: Messages aren't necessarily received in the same order they were sent.
- **One non-FIFO queue per source**: Each process p has a single non-FIFO queue for each process which p can receive message from.

Race condition is intuitively defined as follows: Events e and f of the same process p are said to be in race if e precedes f in bMSC (if e and f are incoming events the corre-

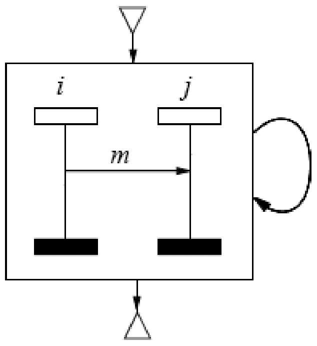

Figure 21: Unbounded HMSC

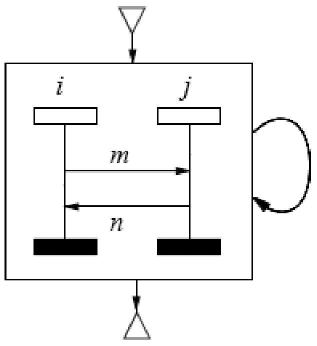

Figure 22: Bounded HMSC

sponding messages must belongs to the same queue too) but semantic of environment doesn't imply that e precedes f.

See Figure 23 for examples (these examples are taken from [3]). In case A and C there aren't any events in race. Case B forces events s and r to be in race for each introduced semantic. In case D single FIFO queue per process and single non-FIFO queue per process force events r_1 and r_2 to be in race. In case E only non-FIFO queues forces events r_1 and r_2 to be in race.

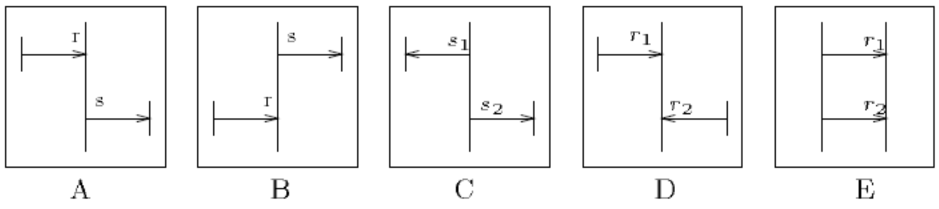

Figure 23: Examples of communication

UBET tool ([29], introduced in [3]) is able to recognize whether two events are in race or not. The tool is even capable to handle coregions (see 2.1.1) in the correct way.

Figure 24 depicts example which is analogous to Example B in Figure 23 except a coregion usage. In this new case events s and r aren't in race in any of the above given semantics.

Authors of the UBET tool introduced—besides race condition and algorithm for detection racing events—**timed MSC** notion. Timed MSC is bMSC whose messages and events' delays at instance axis are labeled by time intervals. This time interval stands for time which message passing can take or how much time instance spends between two events.

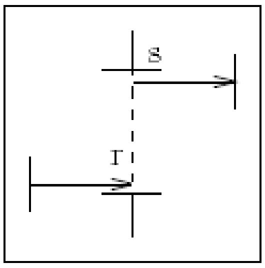

Figure 24: Coregion example

These intervals may introduce some kind of time conflicts. These conflicts may cause that modeled run needn't to be executable in proposed time intervals. For details about related work please see [3].

4 Other MSC Checking

In this section, we provide several properties of MSC. Contrary to the previous section, we focus on language (semantic) features as MSC membership, model checking and realizability. For different MSC language properties not mentioned here, see [8, 7].

4.1 Simplified MSC Model

MSC (as it is specified in Section 2) is a very complex model with many constructs and attributes. Because proving MSC's theoretical properties would be much harder if the complete model had been used, simplified model of MSC is defined. As is commonly given by theoretical publications, **sMSC** (simplified bMSC) is an bMSC without coregions, timers, lost and found messages, creation and termination of processes, compositions and inline expressions (e.g. loops, alternations).

sMSC is not sufficient for specification of real systems, e.g. due to absence of loops. For these reasons, MSC graphs are provided. Consequently, this section is focused on MSC graphs. sMSC properties and algorithms are provided in [1].

4.1.1 MSC graphs

Graphs with sMSCs in their nodes are called **MSC graphs**. Each MSC graph has a distinguished initial vertex and a set of terminal vertices. Paths starting at initial vertex and ending at some of terminal vertices are denoted as **accepting paths**.

To know how MSC graph specifies a set of sMSCs, concatenation of two sMSCs has to be defined. (Asynchronous) concatenation corresponds to natural process-by-process pasting of two sMSCs together (the same as vertical operator in Subsection 2.1). Formal definition can be found in [4]. An MSC (a described communication sequence) of an MSC graph G is then specified as a concatenation of sMSCs of all nodes on an accepting path of G. For a given MSC graph G, a set of all MSCs of G is called a **language** $L(G)$ (see [2] for details).

4.1.2 Implied scenarios

MSC describes global system behavior. Local agents must be implemented separately and all their implementations together make up distributed implementation of system. However, distributed implementation can exhibit some behaviors not specified by MSC. We call such behaviors **implied scenarios**.

We denote as $L(G)$ a language of an MSC graph G, usually specified by developer. Let every process of system be specified by a finite automaton with an alphabet of events (sending and receiving messages). Further, let these automata run concurrently. We denote as $L^w(G)$ language of all concurrent runs respecting following conditions: precedence of message send event to corresponding receive event and FIFO assumption between each couple of processes. The set of implied scenarios is the difference between these languages, $L^w(G) \setminus L(G)$. Then, we say $L(G)$ weakly implies $L^w(G)$ and denote $L^w(G)$ as **weak closure of** $L(G)$.

4.2 MSC Membership

sMSC can be used to specify both desirable and undesirable behavior of a distributed system. In these situations, it is important to know whether given sMSC M is in set of sMSCs specified by MSC graph G. This property is called MSC membership. The hardness of MSC membership verification depends on the semantics of the MSC graph G.

First, let us consider $L(G)$ to be the semantics of G. Then there is an algorithm for checking whether $M \in L(G)$ in $\mathcal{O}(|G||M|^k)$ time, where $|G|$ is the number of vertices of G, $|M|$ is the number of events in M and k is the number of processes of M. Moreover, the problem is proven to be NP-complete with respect to $|G|$ and $|M|$.

For $L^w(G)$ semantics of MSC-graph, the problem is much easier. An algorithm in $\mathcal{O}(|G||M|)$ time is given in [2]. Because real systems usually exhibit behavior corresponding to $L^w(G)$, we consider result on $L^w(G)$ to be more important for practical utilization of MSC membership verification.

4.3 MSC Graph Boundedness

Boundedness is defined in Subsection 3.6. It is a determinant for decidability of other properties that are more important in practice. If decidability of individual property is not determined by boundedness, then complexity of corresponding deciding algorithms usually is.

The following implications of MSC graph boundedness seem to be the most significant ones:

- Number of messages in a network is bounded.
- All processes stay roughly synchronized.

In an unbounded graph, some process or processes can do infinitely many more steps than the others. Such an execution is called **process divergence**.

The main formal language difference between bounded and unbounded MSC graph is the regularity of a language. While language $L(G_b)$ of a bounded MSC graph G_b is known to be regular, language of unbounded graph is not generally regular.

Determination of graph boundedness (for k processes and upper bound m for events of sMSC at every vertex) can be done in $\mathcal{O}(|G| * m * 2^k)$ time and is coNP-complete. See [4] for the proof.

Process divergence can be detected using syntactic checks and has been implemented in MESA tool [22]. The algorithm has been provided in [5].

4.4 MSC Model Checking

Distributed systems are required to satisfy various conditions. Let us recall that $L^w(G)$ denotes the weakly implied language of MSC graph G. Requirements on the system are specified by an automaton A over the alphabet Σ of messages. This automaton accepts undesirable executions, represented by language $L(A)$. Model checking of a given property is then a problem of determining emptiness of $L^w(G) \cap L(A)$, i.e. there are no undesired executions in $L^w(G)$.

4.4.1 Asynchronous concatenation

Generally, asynchronous concatenation (see 4.1.1) makes the model checking problem undecidable with an exception for bounded graphs. For an automaton A and a bounded MSC graph G of $|G|$ vertices, k processes and at most m sMSC events at every vertex, the problem can be solved in $\mathcal{O}(|A| * 2^{k*|G|} * (|G| * m * k)^k)$ time and is in PSPACE. For details see [4].

4.4.2 Synchronous concatenation

sMSCs in MSC graphs can be concatenated in other way then specified in 4.1.1. Synchronous concatenation requires all events of the previous sMSC to be executed before any event of the following sMSC starts. For precise definition, see [4].

Model checking problem is proven to be coNP-complete under synchronous concatenation [4].

4.5 Realizability

As already mentioned in 4.1.2, distributed implementation of a system specified by sMSC or MSC graph can exhibit behaviors not specified by that MSC (graph). $L(G)$ and $L^w(G)$ denote languages of specification and implementation respectively. MSC graph G is **weakly realizable** iff $L(G) = L^w(G)$.

Sometimes distributed implementation of weakly realizable set of MSCs can get to a deadlock state. Set of sMSCs (and specifying MSC graph) is considered to be **safely realizable** iff it is weakly realizable and distributed implementation can't reach a deadlock state.

Notion of weak realizability seems not to have practical usability, because it is easier to determine safe realizability than the weak one. Therefore, results on decidability of weak realizability were not provided.

For finite set of sMSCs (specified by acyclic MSC graph), the problem of safe realizability is in P-time. On bounded MSC graph, the problem is in EXPSPACE and PSPACE-hard. Unbounded MSC graph makes it undecidable. For detailed description and proofs see [2].

Since MSC is model for scenario-based system design, single processes have to be modeled by another means of model-based design, e.g., transition machines, automata or SDL. (Safe) realizability is the crucial property for convertibility of MSC graphs to any such model. More details on this conversion will be provided further in the text.

In some branches of MSC graph, different alternatives of communication can correspond to send events of different system processes. Then, the decision among the alternatives doesn't depend on one local process and deadlock can occur. If the system is safely realizable, choice in every branch of MSC graph is local. The algorithm for non-local choice problem is given in [5] and implemented in MESA tool ([22]).

5 Synthesis

MSC is a suited form of specification at almost top most level of abstraction. During implementation of described system, developers at each level of development process gradually refine the system design up to the particular implementation. Synthesis from MSC design (if feasible) into component based model is therefore natural requirement because it gets design closer to implementation.

Synthesized model is usually a set of finite state automatons (components of system) whose states identify states of implemented components and transitions denote possible actions that the component is able to execute with respect to MSC specification.

There are many notations (specification languages) which describe the automatons e.g. SDL, Uppaal notation, Promela and many others. The aim of these notations, besides role of modeling language, is to formally handle system description which gives the designers certain advantages.

One of the advantages is the ability to formally check some properties which the described system should satisfy. These are e.g. deadlock and cycle occurrence and/or reachability of certain state.

This ability highly improves capability of system designers. They are not only dependent on their own capabilities but an automatic checkers and verifiers can help them to avoid some system defects in early stage of development process.

5.1 LTSA-MSC

LTSA-MSC ([20], an MSC plugin for Labeled Transition System Analyzer [19]) is a typical example of tool which is able to synthesize MSC graph into some kind of state-based model. It uses Labeled Transition System (LTS) as its target specification language of synthesis.

Processing synthesized model the tool is able to reveal some type of gaps in MSC graph specification–it detects implied scenarios. Since LTSA is based on blocking both send and receive events, the only possible kind of implied scenarios detected is non-local choice.

Tool incrementally checks model and tries to find an implied scenario. Each implied scenario is proposed to designer. It's up to designer to figure out whether this scenario represents system defect or it is intended behavior.

6 Introduction to SDL

6.1 Motivation for SDL

Virtually all distributed systems have some characteristics in common:

- they are based on distinguishable entities (servers in the Internet, processes, internal components of a network, etc.)
- there is no global knowledge of the system
- information about the system can be gained only by communication with other entities (no globally accessible variables)
- this communication is usually done by "message exchange"

SDL was created as to describe and specify such distributed systems. It is therefore based on similar characteristics. Individual entities of the global system are denoted as **processes**.

Description of a system

A communication system specified by SDL is composed of **processes** that are administered concurrently.

Behavior of each **process** is specified by its **Finite State Machine**, where the transitions between individual states are sequences of actions (transmission of a message, manipulation with the internal variables, etc.). These sequences of actions are **indivisible** – once started, the whole transition has to be finished (it is impossible for a process to be stopped in between two states). See Figure 25 for a short example.

States are therefore places, where the process waits for an event and transitions are reactions to accepted events.

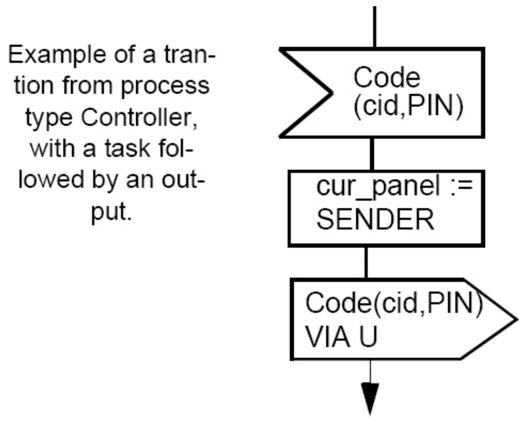

Figure 25: Finite State Machine

6.2 Processes

Processes can have inner structures – **subprocesses** and **variables**. These are visible inside the process and its substructure. Collisions on variables are avoided by interleaving the subprocesses. That is, in each moment a maximum of one transition is done at the same time. Figure 26 shows an example of process specification in SDL.

Example

SDL structure can be shown nicely on networking protocols – processes are individual machines on the network (and therefore are working independently on each other). Their inner structure (daemons, applications, etc.) is modeled by subprocesses. These subprocesses have access to shared files and other types of data on the machine.

6.3 Communication Between Processes

Exchange of messages is the only way of communication in a system modeled by SDL. Message sending can occur only as an action inside a transition.

It is assumed in SDL that communication is operating over a reliable layer therefore messages cannot overtake each other, do not get lost, and their integrity is preserved. If not stated otherwise, messages are delivered instantly. It is however possible to declare that a specific set of messages should be **delayed** or **prioritized**.

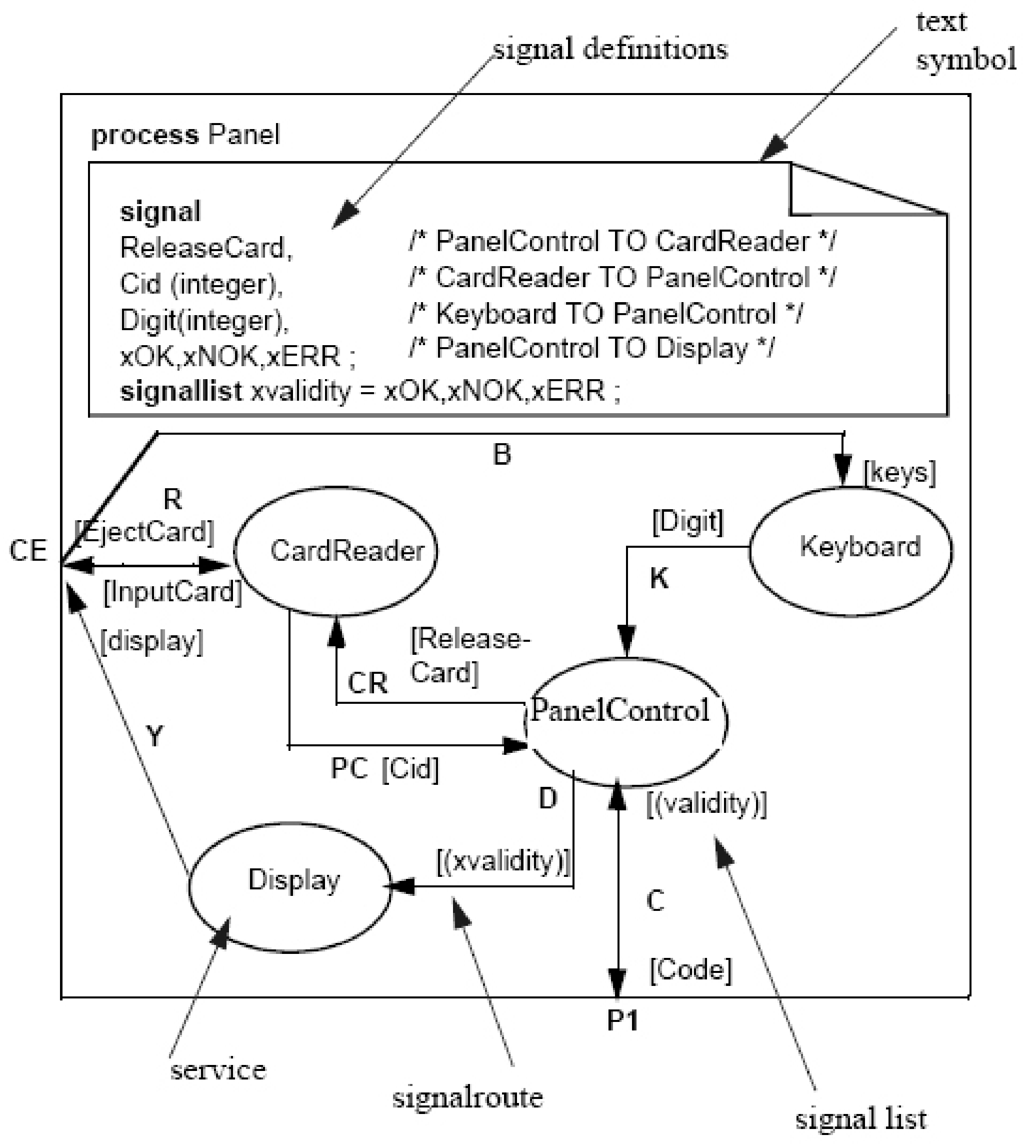

Figure 26: Process

Received messages handling

Each process has exactly one **input port** that behaves like a queue. Incoming messages are put into this input port accordingly to their arrival time. In the simplest case, the oldest message in the input port is processed (according to the message contents the appropriate transition is selected) and the message is then deleted.

The simplest method of message processing shown above can be altered in these ways:

- each state can specify types of messages it wants to process (other types are ignored and left in the queue for other states)

- if a prioritized messages are inside the input port, they are checked before all of the others
- it is possible for a process to change states even without message consumption

6.4 Conclusion

SDL can be used to effectively model distributed systems similar to communicating protocols. It is based on interaction of components of known inner structure and from these interactions the global system behavior emerges. This approach is significantly different to that of MSC in which the system is modeled from a global view by means of concrete message exchanges (without any knowledge of how the individual components work).

It is therefore possible to use SDL for specification of different distributed systems as well as verification of a modeled system behavior.

7 Deeper Exploration of SDL

7.1 Introduction

SDL will be described in this section in a way specified by **ITU-T Recommendation Z.100** [14]. We will be describing SDL in greater detail than the previous section, but the main aim is still explanation of basic principles than a precise syntax description (for these occasions please consult [14]).

It is important to emphasize that some of the software products do not use full strength of SDL, or their version of SDL is not completely equivalent to the SDL specification.

Basic structure of SDL

As we've shown already in the previous section, SDL is a language developed for specification and behavior description of distributed real-time systems. Two different approaches can be used for modeling in SDL – graphic and text syntax. Both are of equal expressive strength.

SDL is based on a reasonably small set of constructs called **primitives** that have precisely defined semantics. All of the other constructs can be unambiguously translated into these primitives and so semantics of the whole SDL is precisely defined. Arbitrary expansion of SDL with unambiguous translation to the SDL language primitives is a correct SDL extension. Unfortunately, most tool will not probably support it.

7.2 SDL Structure - Syntax

Basic concepts

A system is modeled by **agents** communicating with each other by asynchronous signal exchange through channels defined beforehand. Agents are controlled by an internal finite state automaton (**transition machine**) with its transitions defined as reactions to received signals. Each agent has exactly one **input port**.

There are three different types of agents:

- **system** – contains the whole modeled system, communicates with the environment, substructure (blocs and processes) are administered concurrently.
- **block** – an analogy to "subsystem" and so substructure (blocs, processes) are administered concurrently
- **process** – an analogy to a functional unit of a system – can contain subprocesses that are administered in parallel (**interleaving** mode)

Process communication

Processes in SDL communicate by exchanging signals through explicitly defined communication channels. A channel can be created between arbitrary agents, in a special case, a process can send messages even to itself. Received signals are added to the input port queue.

Channels can be one or bi-directional, they are reliable (messages cannot be lost in the channel and cannot overtake one another within the same channel), channels may or may not have delay between send and receive actions. More than one channel can be created between two agents.

7.3 Semantics of SDL

Process behavior

Every process is controlled by its **transition machine** (see Figure 27) and is always in one of these conditions:

- process is waiting in a state for an event (that has an transition assigned to it in this state)
- process is in the middle of a **transition** (possibly leading to the same state) – **transition** is a finite sequence of actions eventually with side-effects (e.g. changing value of individual variables, sending of a signal, etc.). The end state of an transition may be determined during the transition depending on variables values, sent signals, etc.

In SDL, every event is modeled as a signal instance consumption from the appropriate input port. The one and only exception is a **spontaneous transition** that is explicitly defined for individual states. This transition type can be carried out indeterminately at any time if the process is in the appropriate state.

Signal consumption from the input port

Signals in the **input port** are ordered by time of arrival. When a signal is consumed, it is deleted from the input port. In the simplest case, the input port works as a FIFO pipe ordered by arrival time.

There can be following exceptions from the simple FIFO rule:

- signals can have a priority flag – all signals with this flag are checked before any non-priority signals.
- each state can explicitly specify a set of signal types it cannot consume (it is called a **safe** set) – these signals are skipped in the input port and left for following states.
- if no signal can be consumed, state can have a **continuous signal** definition that is administered instead of a standard signal.

Remote procedures

Individual processes can make their procedures available to other processes. A **remote procedure** call is a signal emission that is processed as any other signal. **Remote procedure** uses variables belonging to the process performing the procedure, the caller is waiting for a reply or a timeout and cannot administer other actions.

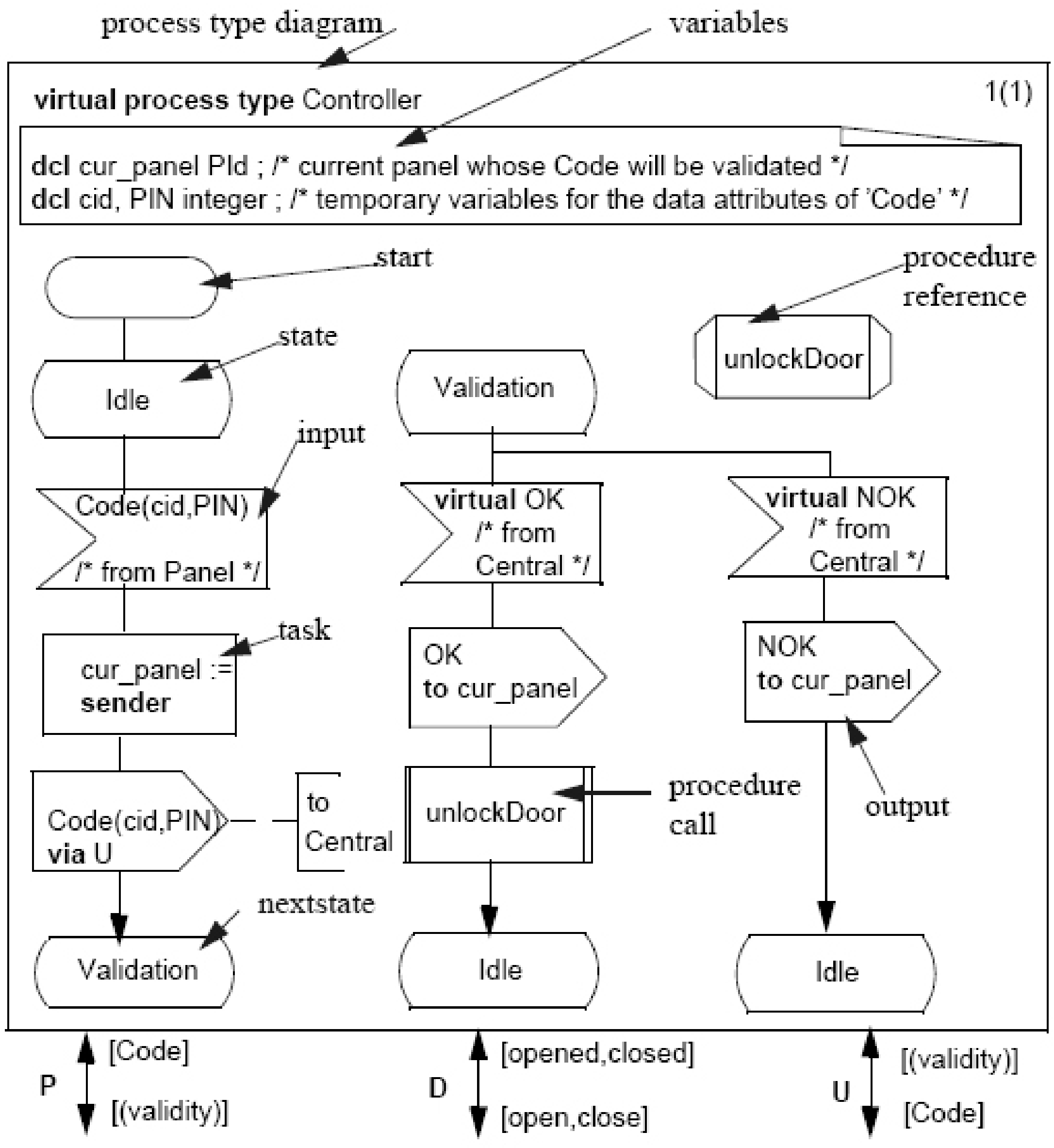

Figure 27: Finite State Machine of a process type

8 MSC2SDL

There is a possibility to generate an SDL specification automatically from an MSC specification. This helps to bridge the gap between the requirement and design phase and the implementation phase. A general approach for synthesis from MSC to SDL was discussed in an article [16]. This approach has been implemented in the MSC2SDL tool that seems to be currently unavailable. The intuitive description of the synthesis is given below.

As already said in the previous section, each SDL process has a single FIFO queue for arriving messages, regardless of the source. Messages sent to a process by different processes are merged into the process's single input queue in the order of their arrivals. In a system specified be MSC their receive order is not given. Rather, the order depends on the underlying architecture and the individual processes interleaving. If not explicitly specified otherwise (by the **safe set** for individual states), SDL instances implicitly discard signals that are in front of their input queue and are not expected at the current state. These discarded signals, which may be required in the next states, may lead to a deadlock.

The approach described in [16] requires besides MSC specification an SDL architecture as input as well. SDL architecture defines set of processes and communication channels between them and has to adhere to the following rules:

- all processes described in the MSC are present in the SDL architecture, and
- for each message m sent from an instance I to an instance J in the MSC, there is a channel ch in the SDL architecture that can convey M from process I to process J.

In order to prevent deadlocks in the SDL specifications, the approach adds an **SDL save construct** for each signal that may arrive in the input queue earlier than expected. The save set is built on the basis of the order relation between the events of the given MSC with reference to the instructions mentioned in [16].

For each MSC process, the tool automatically generates the corresponding SDL process. MSC constructs are translated into SDL constructs on a one-to-one basis without any intermediary representation. The approach inserts a new SDL state before each MSC input event with a save construct for calculated message types.

Commercial tools for synthesizing SDL from MSC (such as **KLOCwork MSC2SDL synthesizer** that is an internal subpart of **Telelogic Tau SDL Suite** [28]) are currently available. Authors of this tool avoided the problem of calculating the save sets and added the save construct for each message type in each SDL state. This approach is not optimal–for certain systems, a deadlock detectable in the MSC specification is not detectable after the synthesis.

Acknowledgements. We would like to thank Petr Gothard and Radek Sedláček for discussions and comments.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[3] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. In *the 2nd International Conference on Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 1996.

[4] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, volume 1664, pages 114–129, London, UK, 1999. Springer-Verlag.

[5] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *the 3rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 1217 of *Lecture Notes in Computer Science*, pages 259–274. Springer-Verlag, 1997.

[6] H. Ben-Abdallah and S. Leue. MESA: Support for scenario-based design of concurrent systems. In *the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 1998.

[7] B. Bollig. *Automata and Logics for Message Sequence Charts*. Thèse de doctorat, Department of Computer Science, RWTH Aachen, Germany, May 2005.

[8] B. Bollig. *Formal Models of Communicating Systems — Languages, Automata, and Monadic Second-Order Logic*. Springer, June 2006.

[9] B. Bollig, C. Kern, M. Schlütter, and V. Stolz. MSCan: A tool for analyzing MSC specifications. In Holger Hermanns and Jens Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 455–458. Springer, March 2006.

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

[11] G. J. Holzmann. Early fault detection tools. In *TACAS'96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1996.

[12] G. J. Holzmann, D. A. Peled, and M. H. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, 2:86–95, 1996.

[13] ITU Telecommunication Standardization Sector. ITU recommandation Z.120, Message Sequence Charts (MSC), 1996.

[14] ITU Telecommunication Standardization Sector - Study group 17. ITU recommandation Z.100, Specification and Description Language (SDL), 2002.

[15] ITU Telecommunication Standardization Sector - Study group 17. ITU recommandation Z.120, Message Sequence Charts (MSC), 2004.

[16] F. Khendek and X. J. Zhang. From MSC to SDL: Overview and an application to the autonomous shuttle transport system. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 228–254. Springer-Verlag, 2005.

[17] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[18] S. Leue and M. Rezai. Synthesizing software architecture descriptions from message sequence chart specifications. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, pages 192–195. IEEE Computer Society, 1998.

[19] LTSA — Labelled Transition System Analyser. `http://www.doc.ic.ac.uk/ltsa/`.

[20] Message Sequence Chart plugin — an extension to the Labelled Transition System Analyser (LTSA). `http://www.doc.ic.ac.uk/ltsa/msc/`.

[21] A. M. Maeda, J. Aoe, and H. Tomabechi. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience (SPE)*, 24(7):603–622, 1994.

[22] MESA — Message Sequence Chart Editor Simulator Analyzer. `http://tele.informatik.uni-freiburg.de/Mesa/`.

[23] MSCan — Message Sequence Charts analyzator. `http://aprove.informatik.rwth-aachen.de/~kern`.

[24] M. Mukund, K. N. Kumar, and M. A. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pages 521–535, London, UK, 2000. Springer-Verlag.

[25] A. Muscholl and D. Peled. Deciding properties of message sequence charts. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 43–65. Springer-Verlag, 2005.

[26] A. Muscholl, D. Peled, and Z. Su. Deciding properties for message sequence charts. In *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 226–242. Springer-Verlag, 1998.

[27] ObjectGeode. `http://www.verilogusa.com/products/geode.htm`.

[28] Telelogic Tau SDL Suite. `http://www.telelogic.com/products/tau/sdl/`.

[29] UBET (formaly named MSC/POGA). `http://cm.bell-labs.com/cm/cs/what/ubet/`.

[30] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool support for behaviour model elaboration using implied scenarios. In *the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003*,

volume 2619 of *Lecture Notes in Computer Science*, pages 597–601. Springer-Verlag, 2003.

[31] UPPAAL — A Tool Suite for Verification of Real-Time Systems. `http://www.brics.dk/BRICS/FormalMethods/UPPALL.html`.