MESSAGE SEQUENCE CHARTS
IN THE SOFTWARE ENGINEERING PROCESS

S. MAUW and M.A. RENIERS and T.A.C. WILLEMSE

Department of Mathematics and Computing Science, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

The software development process benefits from the use of Message Sequence Charts (MSC), which is a graphical language for displaying the interaction behaviour of a system. We describe canonical applications of MSC independent of any software development methodology. We illustrate the use of MSC with a case study: the *Meeting Scheduler*.

Keywords: Message Sequence Chart, software engineering process, groupware.

1. Introduction

The common agreement is that software engineering is a difficult discipline. Despite the methodologies that describe the partitioning of the software engineering trajectory into phases including the deliverables for each phase and techniques that can be applied in these phases, a great number of industrial software engineering projects encounter unanticipated problems. Unfortunately, pinpointing the exact causes for these problems is not always possible, but there are a few well known issues that give rise to these problems. Among these issues are the shifts between subsequent phases and version-management of documentation and software, but also the more basic communication problems between the client and the engineering team.

The language *Message Sequence Chart* (MSC) is a graphical language, initially developed to support the SDL methodology for describing possible scenarios of systems and is standardised by the ITU. In the past decade, many features have been added to the core language. This culminated in the documentation for the most recent version, MSC 2000 [1], describing its syntax, semantics and its conventions.

Traditionally, MSC has been used in the area of telecom oriented applications. There, it has earned its medals for visualising and validating dynamic behaviour (see the SDL Forum proceedings [2, 3, 4, 5, 6]). However, over the past years, alongside the increased expressiveness of the language also the specifying of dynamic behaviour has become a major topic of research and practice. Being a standardised language, one of the main advantages of MSC over comparable languages is that is has been formalised. Moreover, the language is understandable both by the specialist and a layman, i.e. it can serve as a medium for communication between groups with different backgrounds. This is particularly useful in the setting of software engineering.

In this paper we will give an overview of the canonical applications of MSC within the software engineering trajectory, without focusing on one particular methodology. This is done by identifying the commonly occurring phases in a number of software engineering methodologies, and explaining the applications of MSC in and between each phase, based upon this identification. Some of these applications are already much used, while other applications are not that straightforward. Wherever possible, references to literature or ongoing research is provided.

In order to present more than an abstract framework, in this paper a relatively trivial case study is presented. Using this case study, various of the discussed applications of the language MSC are shown in practice, thereby providing a more profound understanding of the canonical applications of MSCs and of the language itself. The case study we will discuss is an application that is part of an *Inter Business Communication Support System* software suite, called the *Meeting Scheduler*.

We will start by introducing the language MSC in a nutshell in Section 2 for the common understanding of the diagrams presented in this paper. The application of MSCs in the software engineering trajectory is subsequently discussed in Section 3. There, the canonical applications in each phase, and between different phases, are presented. Using the Meeting Scheduler as a running example, in Section 4, some of the canonical uses of MSC are presented, thus providing a concrete example of both the applications of MSC and the language itself. At the end of this paper, in Section 5, some concluding remarks are made.

2. Message Sequence Charts

MSC (Message Sequence Charts) is a graphical specification language standardised by the ITU (International Telecommunication Union). In this section we will give an overview of the main features of the MSC language. For a more detailed introduction the reader may consult [7, 8].

MSC is a member of a large class of similar drawing techniques which more or less independently arose in different application areas, such as object-oriented design, real-time design, simulation and testing methodology.

The main virtue of these languages is their intuitive nature. Basically, an MSC describes the communication behaviour of a number of logically or physically distributed entities, displaying the order in which messages are exchanged. Graphically, the life-line of an entity is represented by a vertical axis, while the messages are drawn as arrows connecting these life-lines. A simple MSC (such as the one in Fig. 1), can be easily understood by a non-trained user, which makes the MSC language very suitable for communication with e.g. clients.

The MSC language as used in this paper stems from the telecommunication world. The popularity of MSC in this area is explained by the fact that typical telecom applications feature distributed reactive systems with real-time demands, for which a scenario based description with MSC is particularly useful. While the application of MSC in the telecom world dates back to the seventies, the first official ITU recommendation was issued in 1992. Since then, the language was maintained actively by an international user community and supported by commercially available design tools (e.g. [9, 10]).

Over the years, the small and informal MSC92 language developed into a powerful and formalised language, of which the current version is called MSC2000 ([1]).

The choice for using MSC2000 in this paper is motivated by these factors: MSC is a formal, standardised and well supported language. Although, in the context of the ITU, MSC is embedded in the SDL design methodology for distributed telecom applications ([11, 12, 13, 14, 15]), this does not impose any restrictions on its use in a different methodological context. We consider MSC as a generally applicable tool which can be used to strengthen the software development process independent of the adopted methodology.

The remainder of this section will be devoted to explaining the main constructs from the MSC language.

2.1. *Basic Message Sequence Charts*

As explained above, a basic MSC consists of vertical axes representing the life-lines of entities and arrows connecting these lines, which represent messages. MSC M from Fig. 1 contains four entities, p, q, r, and s (In this section we will introduce MSC with meaningless abc-examples. More useful MSCs are given when discussing the case study in Section 4). Instance p first sends message a to instance q, which subsequently receives this message. Messages in an MSC are considered asynchronous, which means that the act of sending a message is separate from the reception of a message. Of course the sending of a message must occur before the reception of this same message, but between these two events, other events may take place. We say that the sending and reception of a message are causally related events.

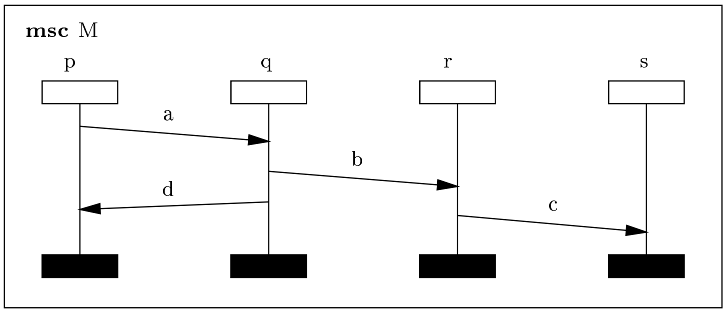

Fig. 1. A basic MSC.

After reception of message a by instance q, instance q will send message b. The reception of a and the sending of b are causally related, because they occur in the given order on the same instance axis. After sending b, we come into a state where two events are enabled: the reception of b and the sending of d. Since in the diagram no causal dependency between these two events is expressed there is no implied order of execution. Continuing this line of reasoning, we find that a basic MSC diagram defines a number of execution orders of simple communication events.

This interpretation is worked out in mathematical detail in the official MSC semantics (see [16, 17, 18, 19, 20]). In this paper we will not pursue the path of formality, but we will restrict ourselves to intuitive explanations.

In Fig. 2, we have extended the simple MSC with additional information. First, we see that the events of sending a and receiving d are vertically connected by a two-way arrow. This means that we have put a time constraint on the occurrence of these two events: the reception of d must occur within 3 time units after the sending of a.

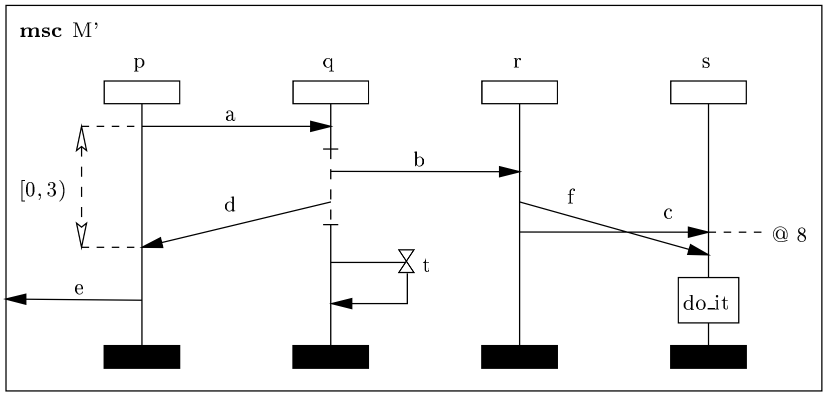

Fig. 2. An extended basic MSC.

Apart from the expression of relative time requirements, MSC also supports the observation of absolute time stamps. This is denoted by the timing attribute connected to the reception of message c. Therefore, this event occurs at time 8.

Next, observe that the life-line of instance q is partly dashed. This means that the events on this part of the instance axis are not causally ordered. The sending of b may occur before or after the sending of d. This allows to reduce determinacy of the specification. This construct is called a *coregion*.

Message e is a special kind of message, namely a message to the environment. Such messages are needed to specify open systems. Message f is added to show that messages are allowed to overlap. This means that there is no a priori assumption about the type of message buffering.

At the end of instance q we have added an example of the use of timers. This example denotes the setting of a timer with name t, followed by the subsequent time-out signal of this timer. It is allowed to detach the time-out event from the setting of the timer. In this case, the hour glass symbol and the attached timer name must be repeated.

Finally, notice the small box at the end of instance s. This stands for a local action, performed by instance s. This is simply an action event of which we know the name (do_it), which must occur after the reception of f.

2.2. *Structured Message Sequence Charts*

Although basic MSCs yield quite clear descriptions of simple scenarios, structuring mechanisms are needed to nicely express more complex behaviour. There are three ways of defining substructure within an MSC: MSC references, instance decomposition and inline expressions (see Fig. 3).

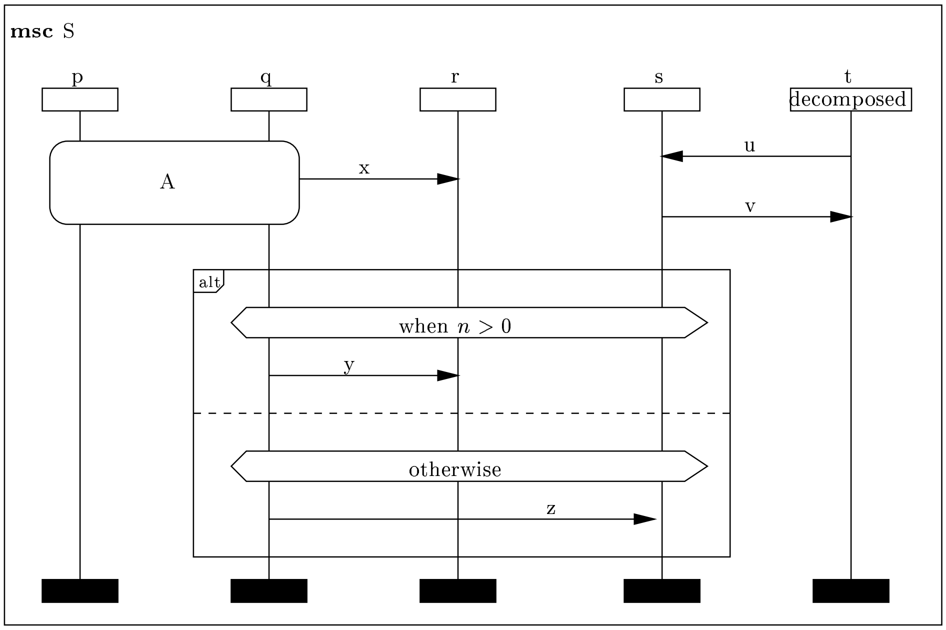

Fig. 3. An MSC with sub-structure.

This example shows a reference to MSC A, which must be defined elsewhere. MSC A is simply thought to replace the area of the MSC reference which covers the instances p and q. The diagram also shows that we expect that a message x is leaving the MSC reference. This implies that within MSC A a message x to the environment must be defined.

Instance decomposition is similar to MSC references. Rather than abstracting from the internals of a region within an MSC, it serves to abstract from the internals of an instance. In the example instance t is labelled as a decomposed instance, which means that the reader must refer to an MSC named t to find the description of the internal behaviour of this instance. MSC t will in general contain a number of (new) instances, which co-operate to obtain the external behaviour of instance t. This clearly implies that MSC t must contain at least a message u sent to the environment and a message v received from the environment.

The third structuring mechanism in Fig. 3 is the *inline expression*. An inline expression consists of a framed region of the MSC with in the upper left corner the name of an operator. The operands to which the operator applies are separated by a dashed horizontal line. In this case, the operator is the alt operator which stands for *alternative*. The two operands which are considered alternatives consist of message y and message z, respectively. In its general appearance, the choice between the alternatives is made non-deterministically. However, by using *conditions* the selection criterion can be made explicit. In this case, the alternatives are preceded by conditions (represented by stretched hexagons) testing the value of some variable n. Please notice that such a condition does not represent a synchronisation of the involved instances. It merely expresses that the instances reach agreement on the continuation, possibly not exactly at the same moment of time.

The conditions as used in this example also hint at the use of data variables in an MSC. Since we do not need data in our examples, we will not discuss this issue in greater detail. A more symbolic way of using conditions is also supported, as shown in Fig. 7. It is allowed to simply label a condition with a symbolic name, which can be asserted and inspected.

In its general appearance an inline expression may contain other operators than the alt operator, such as loop to express repetition and par to describe (interleaved) parallelism. The allowed number of operands depends upon the operator used.

2.3. *High-level Message Sequence Charts*

A different construct which supports modularisation of MSC specifications is a High-level MSC (HMSC). An HMSC serves as a kind of road-map linking the MSCs together. In Fig. 4 we see the relation between three MSC references, A, B, and C. The upside down triangle indicates the start point. Then, following the arrow we arrive at a condition, which gives a hint about the state the system is in initially (idle). Then, we encounter the first MSC to be executed, MSC A. After executing A there is a choice between continuation: B, preceded by the condition ok, and C, preceded by condition retry. After selecting the left branch, B is executed which is followed by another triangle, which indicates the end of the HMSC. If we would have selected the right branch, MSC C is executed, after which we restart at MSC A.

2.4. *Additional MSC constructs*

Until now we have discussed all MSC language constructs needed to understand the remainder of this paper. There are some more useful constructs, but we will only mention these briefly.

An *MSC document* is a drawing which can be seen as the declaration of a coher-

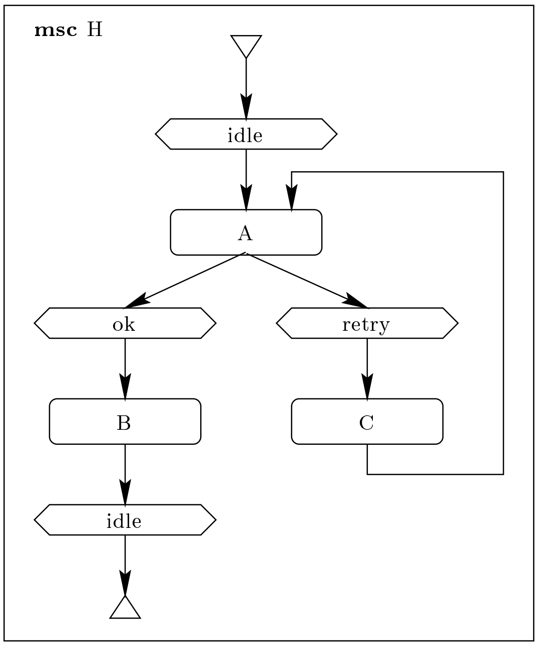

Fig. 4. A High-level MSC.

ent collection of MSCs, instances, variables and other objects. In an MSC document a distinction is made between *public* and *private* MSCs as to control visibility to the outside world. Also, the decomposition hierarchy which emerges when using the decomposition construct iteratively is reflected in the use of MSC documents.

Finally, we mention special syntax for expressing a more rigid order on the contained events, for the creation and stopping of instances, for describing method calls and replies, and for defining messages that do not arrive at their destination.

3. The application of MSCs in software engineering

In this section we first present a simplified view of the software engineering process. Later this view will be used to connect the applications of MSC to.

3.1. *Software engineering*

There are many models that describe the software engineering process. We mention the waterfall model [21], the incremental delivery model [22], the spiral model [23], the V-model [24], and the cluster model [25]. In general these models prescribe the same types of activity, but differ in the way these activities are partitioned into phases, the order in which the phases are executed, and the deliverables. We will not focus on one of these models specifically. Instead, we will pay attention to a number of frequently occurring phases in these models. These are requirements engineering, specification, design, and implementation. Summarising, these phases can be characterised as follows.

In the *requirements engineering* phase it is clarified what the system is supposed to do and in which way it is dependent on the environment of the system. This not only refers to the functional requirements the system should satisfy, but also includes non-functional requirements like timeliness, dependability, fault-tolerance, etc.

In the *specification* phase the user requirements are analysed and a set of software requirements is produced that is as complete, consistent and correct as possible. In contrast with the user requirements, the software requirements are the developer's view of the system and not the user's view. The result of this phase is a specification of the system in natural language, a formal specification language, or possibly a combination of both.

In the *design* phase decisions are taken as to the partitioning of the system into subsystems and interfaces with a well-understood and well-specified behaviour. Also the interaction of the subsystems is considered carefully. The design will serve as a blueprint for the structure of the implementation.

In the *implementation* phase the design from the design phase is realised in terms of software and hardware. Typical validation activities are acceptance, conformance and integration testing.

In each of the abovementioned phases verification and validation activities are performed. These activities are intended to verify the results of a phase with respect to the results of other phases (or with respect to requirements not mentioned before). We will not make any assumptions about the order in which phases are executed, the overlapping of phases, or the number of iterations. Based on the distinction of phases, discussed in this section, the use of MSC will be described in the next section.

3.2. *MSCs in the software engineering phases*

Thus far, we have mentioned some frequently occurring phases in the software engineering process. Next, we will discuss the use of MSCs in each of these phases and in the relation between the phases. An overview is given in Fig. 5. The details of this figure will be explained in the course of this section.

3.2.1. *Requirements engineering*

In the requirements engineering phase of the software engineering process we consider two tasks in more detail. These are *requirements capturing* and *requirements analysis.*

The objective of requirements capturing is to obtain a view of the clients wishes. Unfortunately, clients are not always clear in what their wishes are; hence, the user requirements are not straightforwardly obtained. Often employed techniques involve interviews, confrontations with prototypes and conversations with the engineering team. Although the experienced requirements engineer is trained in abstraction

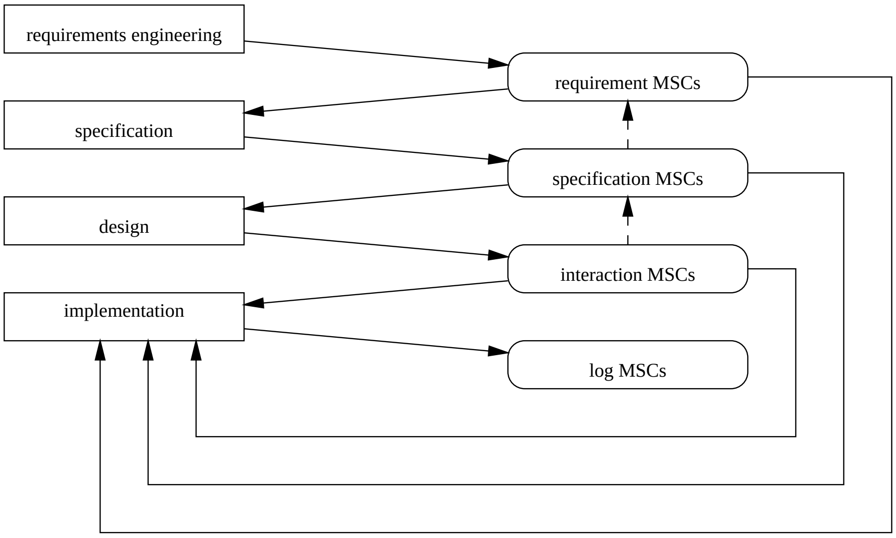

Fig. 5. Overview of the use of MSC in the software engineering process.

and deduction, still, tools are necessary for documenting requirements in a clear and concise manner. Message Sequence Charts can very much assist the process of converting informal documentation into more formal requirements; moreover, MSC eases communications with clients in which harder to understand parts of system's behaviour and implications of a combination of requirements are discussed.

Basically, in every interview with a client, various causal relations can be read. From these relations, one can derive scenarios or use cases, describing parts of the desired system's behaviour. Such a use case describes (part of) the external behaviour of the system placed in its environment. The descriptions can include resource constraints, timeliness constraints, performance constraints, etc. In this paper, we will assume that the result of the requirements capturing phase among others consists of a set of use cases.

The language MSC can be used to clarify use cases in which one or more actors and the system are involved. The roles that appear in use cases are represented by instances in MSCs. Also the system is represented by an instance. MSCs are suited for this purpose as they emphasise the interaction between instances. The interactions between the roles and the system are described by means of messages. Conditional behaviour can be expressed by means of conditions and alternatives.

Consider the user requirement that the system must react within 15 seconds on a request from an initiator by means of an acknowledgement. In Fig. 6 the corresponding MSC is given.

Scenarios are not always considered to be *true requirements*, as they describe the system's behaviour in a very operational manner, possibly containing redundancy. However, the skilled requirements engineer is capable of turning these scenarios into real requirements by abstraction, deduction and combination. This process is called

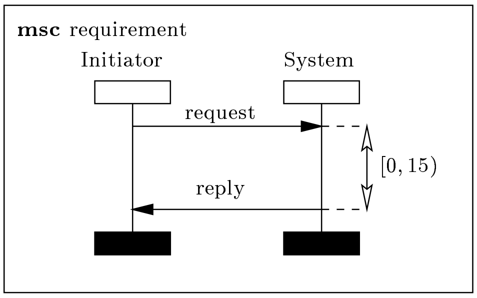

Fig. 6. A simple user requirement.

requirements analysis.

MSCs can be of use in the requirements analysis phase by aiding communication between the engineering team and the client. By the mere task of collecting all user requirements and combining them, system behaviour, foreseen or not foreseen by the client can be derived. Being of a more complex nature than simple user requirements, these composed behaviours are often hard to explain to the client. The concepts of the language MSC can be employed to visualise these more complex behaviours. In this way, communication between the engineering team and the client is eased.

If use cases described by MSCs tend to be large and have overlapping parts, re-occurring parts can be isolated in separate MSCs and be referred to by means of MSC references. The relation between the auxiliary MSCs obtained in this way can be defined in an HMSC. The MSC document allows the separation of *defining MSCs* from *auxiliary MSCs*. Especially in an incremental or iterative software engineering process, the MSC document enables to maintain a good overall view of the MSCs and their relations.

Although MSC does not really add new ways to finding requirements, the benefit of using MSC may be clear: abstracting and deducting information is eased by the overview that is achieved by explicitly focusing on the causal relationships that would otherwise remain hidden in text. References to the use of MSC for use case description are [26], and [27].

In the verification and validation part of the implementation phase the MSCs produced in the requirements engineering phase can be used as descriptions of the test purposes or test cases for acceptance testing.

3.2.2. *Specification*

The specification of the system is not necessarily described by means of formal methods. Often prototypes are built, only parts of the system are described by means of formal methods, or even only natural language is used. The MSCs derived in the requirements engineering phase can be used to serve as the basis for writing a more complete specification of the system. In theory, MSC can also be used for writing specifications. In the literature several papers deal with the generation of a formal specification from a set of (requirement) MSCs: in [28, 29, 30] SDL descriptions are generated, in [31] statecharts are generated, and in [32] ROOM models are generated.

However, here we will focus on the use of MSC for visualising traces, or *runs* of the system. If a specification is developed without using the requirement MSCs and the formalism used for the specification is executable, then the specification can be used to generate specification MSCs. If the language used is less formal, still, it might be possible to extract MSCs based on informal reasoning and a good understanding of the specification. If a prototype of the system is developed, MSCs can be obtained from logging and interpreting execution traces of the prototype. In [33], MSCs are used to visualise the execution sequences that result from partial order simulations of SDL descriptions. In several commercially available SDL tools [9, 10], simulation runs of SDL descriptions are represented by MSCs.

MSCs that result from the specification in the ways described above are useful for comparing the specification with the user requirements. At the right level of abstraction each of the MSCs representing a user requirement should be contained in the MSCs obtained from the specification. Alternatively, the MSCs that represent the user requirements can be used as a monitor for executable specifications such as Promela programs in the Spin tool [34] and SDL specifications in the SDT tool [9].

The specification MSCs can also be used for conformance testing in the verification and validation part of the implementation phase. More details about this use of MSC are given later.

3.2.3. *Design*

The activities carried out in the design phase must lead to a physical and/or logical decomposition of the system into interacting subsystems in such a way that the external behaviour of this collection of subsystems "implements" the specification. As a consequence, the interaction between the subsystems must be specified in a clear and unambiguous way. Message Sequence Charts are especially useful in the description of the interactions in the form of communication protocols, method calls and procedure invocations.

If a physical decomposition of the system is envisioned, the relation between the system and the subsystems is represented in MSC by means of instance refinement (decomposition). In logical decompositions the relation between the different MSCs can be made clear in an HMSC.

As in the specification phase, based on the specification of the subsystems and the interactions between these, MSCs can be generated. These MSCs then also display the internal events. After abstraction from these internal events the resulting MSC must be consistent with the specification MSCs. Hence, the MSCs from the specification phase and the design phase can be compared in order to validate the design with respect to the specification. Since the language MSC is formal, this comparison can also be formalised.

MSCs describing forms of interaction can later be used for integration testing. If the interaction between system components is based on buffering messages, it is possible to determine if this interaction can be realised with a given communication model [35].

3.2.4. *Implementation*

The implementation phase amounts to the realisation of the design in terms of hardware and executable software. Message Sequence Charts can be used in this phase to log execution traces of the implementation. If performance is of relevance, typically all events in such MSCs have a time stamp. In Fig. 12 an example of such an execution MSC is given.

These traces can be inspected manually for unexpected situations or can be compared with Message Sequence Charts defined earlier in the software engineering process. For example, after applying the appropriate abstractions it is useful to compare the traces to MSCs generated by the specification (if any), or to the MSCs issued in the requirements engineering phase.

If errors are detected in the implementation the MSC that logs the trace leading to the error can be used to locate the error in the implementation.

In the verification and validation part of the implementation phase, by means of acceptance, conformance and integration testing the confidence in the systems performance (both functional and non-functional) is validated against the user requirements, the specification and the design, respectively. We explain the use of MSC in conformance testing in some detail. The use of MSC in acceptance and integration testing is similar.

In conformance testing, the behaviour of the implementation is validated against the expected behaviour as described in the specification. In the literature several authors have indicated that the use of MSCs in conformance testing is valuable [36, 37, 38, 39, 40]. In conformance testing the expected behaviour, in terms of observable events of the implementation, is described in a test suite, i.e. a set of test cases. A test case describes a tree of observable events and to each path in the tree it assigns a verdict which specifies whether the described behaviour is correct or incorrect. Execution of a test case results in feeding the implementation with inputs and observing the generated observable events. This execution sequence of the implementation is then compared with the test case. The verdict of the corresponding path in the test tree is the outcome of the test execution.

The use of MSC for the identification of test purposes is advocated by the method SaMsTaG [41, 42, 43, 44]. In the SaMsTaG method a complete test case can be generated from a system specification in SDL and a test purpose description in MSC. The test case is described in the Tree and Tabular Combined Notation (TTCN) [45]. A similar approach is followed by the HARPO toolkit [46, 47].

Among others the papers [48, 49, 50] use MSC for the description of test cases. In [51] synchronous sequence charts, i.e. Interworkings [52], are used for this purpose.

4. Case: The Meeting Scheduler

We will illustrate the use of Message Sequence Charts with a simple case study, baptised *The Meeting Scheduler*. This is an internet application which supports the scheduling of a meeting. In this section we give an explanation of the Meeting Scheduler, but before doing so, we will give the context of its use.

4.1. *Communication support*

The Meeting Scheduler is part of a software suite that supports the communication between people of different enterprises (an *Inter Business Communication Support System*, IBCSS). The main difference with existing packages, such as ERP (Enterprise Resource Planning) packages and business support systems such as Outlook, is that IBCSS focuses on the communication between different enterprises. This reflects current trends in business operation, such as lean production and concentration on core business. The consequence of this development is that production is no longer performed mainly within one enterprise, but within a cooperation of several independent enterprises. Each of these enterprises contribute their share to the final product. The clear cut distinction between customer and producer becomes ever more blurred; both consumer and producer cooperate to achieve a common goal. As a consequence, the spectrum of communication shifts from the intra-business perspective to the inter-business perspective.

Current communication support tools are often not suited to support the inter-business communication process. For instance, these tools assume that every user has the same software environment. It is evident that inter-business support tools must be based on established internet technology, such as web browsers.

An example of such an internet based application is a blackboard system where users can share and manipulate electronic documents (such as the BSCW server [53], which allows access via normal web browser software). Other tools one could imagine are project management tools taking care of e.g. resource planning and decision support systems.

A very simple example of such a communication support system is the aforementioned *Meeting Scheduler*, which we have chosen to demonstrate the use of MSCs on.

4.2. *Informal description*

Scheduling a meeting can be a rather time consuming activity. Dependent on how many people are involved, a number of telephone calls or e-mails are necessary in order to come to a date and time that is convenient to all, or at least to the majority of the participants. The Meeting Scheduler is tailor-made to support the administration of relevant information and communication with the intended participants.

The Meeting Scheduler runs on some internet server and people communicate with the server via e-mail, simple web pages and web forms. The working of the

Meeting Scheduler is best explained by giving the basic scenario of usage.

Two roles can be distinguished: the *initiator* of the meeting and the *invitees*. The initiator takes the initiative of setting up the meeting. He provides the system with the initial information, such as purpose of the meeting, the list of invitees and the list of possible dates and times. Next, the Meeting Scheduler informs the invitees about the meeting and collects information from the participants with respect to the suitability of the proposed dates. If all participants have provided their information (or if some deadline is met), the system reports back to the initiator and suggests the best possible date. After confirmation by the initiator, the final invitation is sent to the participants.

This very basic description can be easily extended with many features. In fact, very advanced tools which support the scheduling of meetings already exist, but these are often platform dependent, and require participants to maintain an on-line agenda.

In the subsequent section some of the uses that are mentioned in Section 3.2 are explained using the Meeting Scheduler. Note that this is not done extensively for all phases. Most notably, no examples are given for the implementation phase. Since the use of MSC for validation is discussed extensively in the literature, only brief remarks are added wherever possible.

4.3. *User requirements*

The techniques for requirements capturing mentioned in Section 3.2.1 can very well be applied to the Meeting Scheduler. For instance, the use of MSC in an interview can be illustrated by transforming the following phrases, taken from an interview, into MSC: "... the initiator feeds the system with the necessary information to send out meeting requests to all potential participants of a certain meeting. These participants should be allowed ample time to respond to these invitations. Eventually, the system will send the current information about potential dates to the initiator who will then decide on a date for the meeting to take place. The system will subsequently inform all participants of the decision of the initiator. Finally, a confirmation of this operation is sent to the initiator..."

The scenario obtained by projecting on the behaviour of the interactions between the initiator and the system is rather straightforwardly deduced from the above sentence (see Fig. 7). Here, the initiator is represented by an instance initiator and the system is represented by a single instance system, thereby portraying the black-box approach. The meeting_info message is used by the initiator to send information vital for the scheduling of the meeting by the system. The message collected_info represents the collected information for the meeting that is communicated between the system and the initiator; the messages decision and confirmation are self-explanatory. The conditions that are introduced can be read as comments, denoting the (required) state of the system.

One can imagine that various scenarios for the Meeting Scheduler describe the causal relationship between the reception of information for a meeting to be sched-

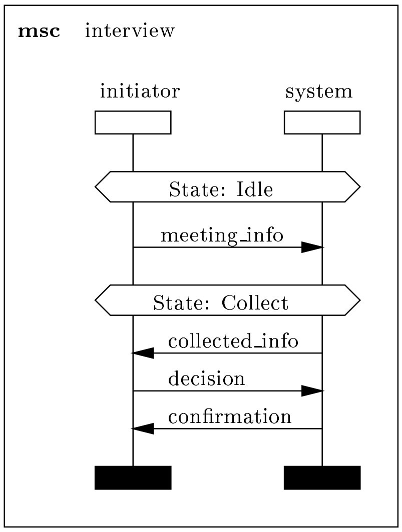

Fig. 7. Scenario deduced from a part of an interview.

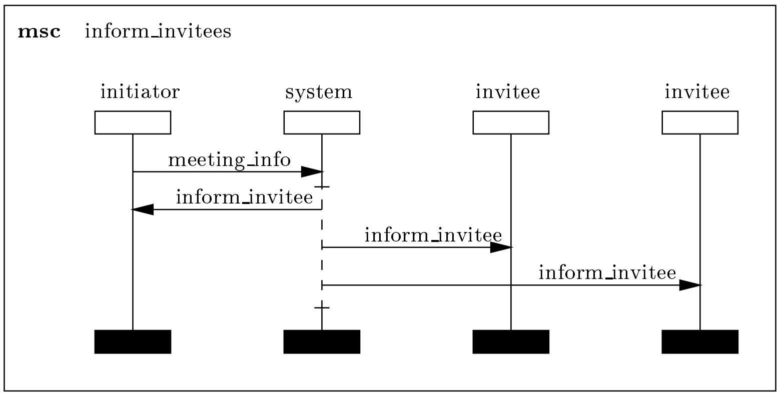

Fig. 8. Requirement deduced from interviews.

uled (denoted by a message meeting_info) and the sending of meeting requests to potential participants of this meeting (denoted by a message inform_invitee). The true (functional) requirement that can be distilled from these scenarios would be one that focuses on exactly that causal relationship (see Fig. 8).

Note that this still is a scenario, and therefore portrays only parts of a system's behaviour. The fact that in this scenario the initiator is also informed about the meeting means that in this case the initiator is himself considered as an invitee, but this is not necessarily always the case.

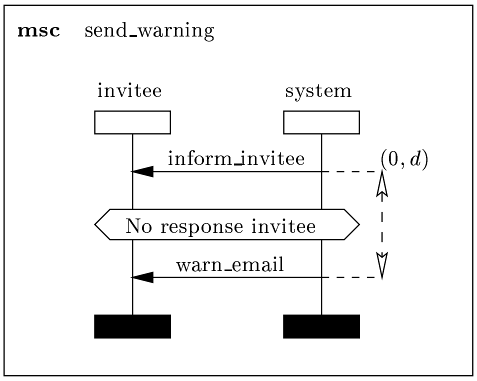

Fig. 9. Requirement deduced from a scenario.

As the discussion in Section 3.1 pointed out, not all requirements can be classified as functional requirements; hence, a language supporting only functional requirements would not suffice. Using MSC, also non-functional requirements, such as the need for time-outs under certain conditions can be illustrated. For example, a non-functional requirement in the Meeting Scheduler would be the sending of a warning message to participants that did not yet respond to the meeting call (denoted by a message warn_email) *before* a deadline (d) is reached. Such a requirement can be elegantly formulated in MSC as Fig. 9 shows.

Thus far, we have focussed on the more trivial user requirements and the scenarios belonging to them. As already mentioned in Section 3.2.1, the combination of requirements may lead to an intricate interplay of causal relations. Finding out these relations already is part of the requirements analysis phase. As an example, a less basic interaction scheme between the initiator and the system for the Meeting Scheduler is considered (see Fig. 10). Overview diagrams such as these assist the communication between the engineering team and the client.

Basically, in Fig. 10 a blueprint for the logical structure for distinguishing between the two options the initiator is confronted with can be read. The information returned by the Meeting Scheduler may or may not be according to the wishes of the initiator. Worst case information may even mean that the invitees for a meeting could not agree on a date for the meeting. Hence, the initiator is confronted with the dilemma of having to decide to cancel the meeting altogether or decide on a date, represented by the MSC reference conclude, or retry to schedule the meeting

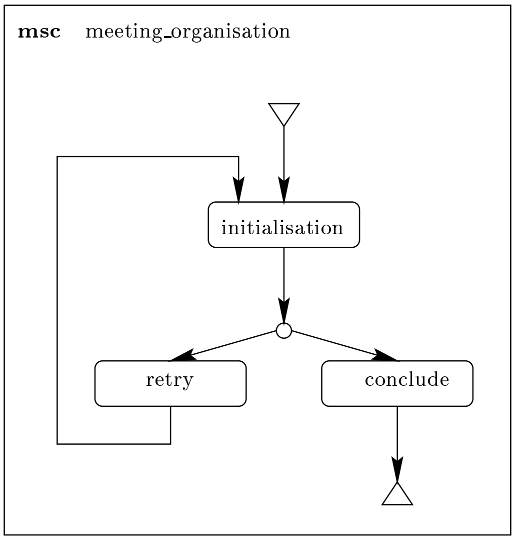

Fig. 10. Combination of user requirements may lead to more complex behaviour.

(possibly using different dates), represented by the MSC reference retry.

The MSC conclude is depicted in Fig. 11; if the invitees could not agree on a date and the initiator decides to cancel the meeting, a cancel message is sent to the system; the system then subsequently responds with a confirmation, using a confirmation message. In case a date is found for the meeting, the system is informed by the initiator about this using a convocate message, and again, the system responds with a confirmation. A similar MSC can be written for the MSC reference retry (not shown here).

Careful comparison of MSC interview (Fig. 7) and the HMSC meeting_organisation (Fig. 10) learns that the MSC interview is one of the possible scenarios described by the HMSC meeting_organisation.

4.4. *Specification*

Although the language MSC can even be utilised for specifying systems, (see Section 3.2.2), we will adopt the language only for validation and visualisation purposes in this phase. Since MSC was also devised for this purpose, we feel it is strongest in this respect. As already mentioned in Section 3.2.2, the ways in which one can obtain scenarios in this phase are plenty; the size and complexity of the Meeting Scheduler would allow for a formal specification, and hence, the generation of traces, or *runs* from this specification is, dependent on the method used, rather straightforward. It would be outside the scope of this paper to give a specification for the Meeting Scheduler, hence, we adopt the operational description of Section 4.2 as a reference for a possible specification for this system.

As an example trace for the Meeting Scheduler, one can think of the scenario depicted in Fig. 12. Basically, this scenario is a combination of various user require-

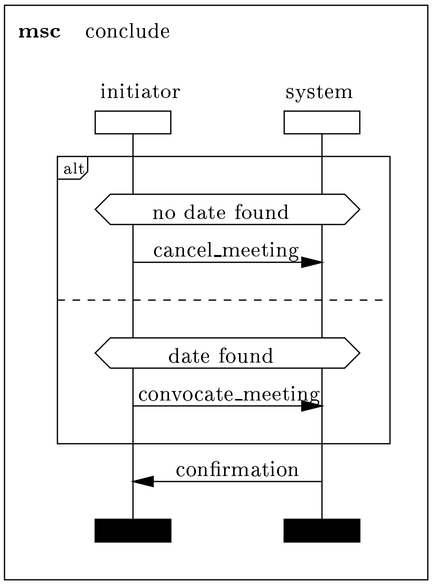

Fig. 11. Part of the complex behaviour.

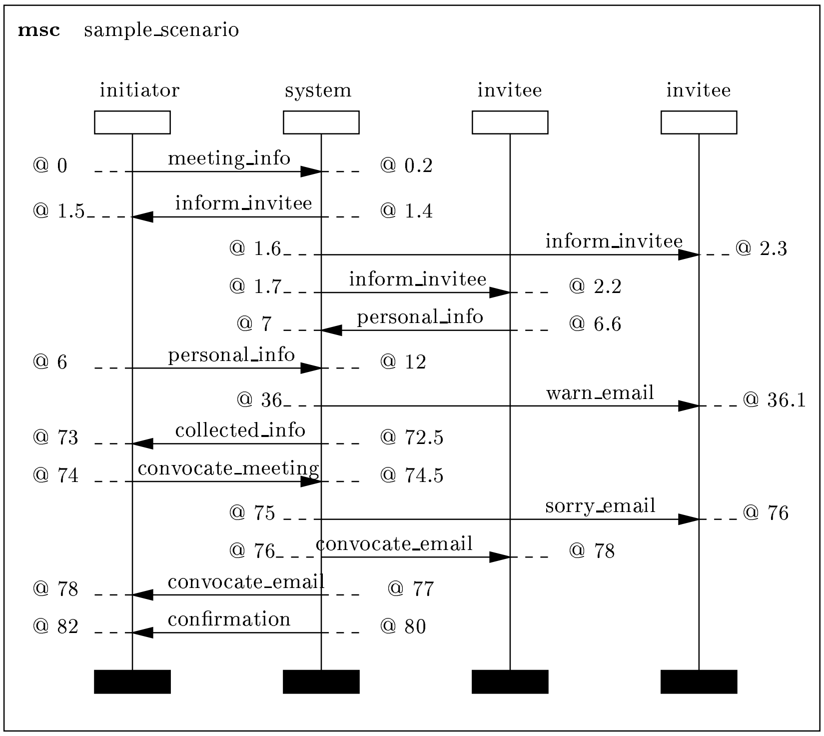

Fig. 12. A typical scenario obtained by a specification.

ments listed in the previous section.

The MSC depicted in Fig. 12 can be validated against the user requirements. For instance, one can observe that the functional and non-functional requirements of Section 4.3 are met. As already mentioned in Section 3.2.2, the scenarios generated in the specification phase are again needed for validating the products of the implementation phase.

4.5. *Design*

The design phase prescribes as one of its main activities the decomposition of the system into subsystems. More concretely, this means that choices have to be made with respect to the desired properties of the system under construction. For the Meeting Scheduler, this boils down to finding logical and/or physical decompositions of the black-box which are chosen in such a way that the requirements of Section 4.2 are fulfilled. Note that the (part of the) design that is discussed here is based on the operational description of the Meeting Scheduler.

The obvious choice for a physical decomposition for the Meeting Scheduler is to consider a decomposition in two subsystems, a front-end and a database. The front-end is a system that deals with the interactions between users of the Meeting Scheduler and as such is the intermediate between the users and the database, whereas the database primarily stores the information posted by users concerning possible dates and times for the meeting.

The interactions between all subsystems involved for the Meeting Scheduler can be grouped, based on a logical decomposition of the global state of the system. The change of state is again, like in Section 4.3, a more complex concept, typically expressed in HMSC (see Fig. 13). Closer observation of Fig. 13 reveals the expected structure of an *initialising*, a *collecting* and a *deciding* phase. In each of these phases, basic MSCs can be used to explain the interactions between various subsystems.

To highlight some of the interactions between the subsystems for the Meeting Scheduler, the MSC references initialise, collect and warn_invitees are highlighted.

The basic MSC initialise (see Fig. 14) describes the essence of which interactions can typically be expected in the initialising phase. Most notably, parts of the interactions identified in the requirements engineering phase (see Figs. 8, 9, 10, 11) reappear in this scenario. Basically, the scenario describes the interaction between the initiator and the front-end of the Meeting Scheduler, in which information for the scheduling of a meeting is communicated using a message meeting_info. Subsequently, the front-end updates the database, and only then it starts sending calls to all invitees for this meeting in random order, (using inform_invitee messages). In order to meet the non-functional requirements (see Fig. 9), two timers are initialised, one for sending a warning message and one for keeping track of the deadline for responding to the invitation.

Collecting information of the invitees is illustrative for typical update interactions occurring between the front-end system and the database, as a reaction to messages from the users of the system (see Fig. 15). Using the message personal_info,

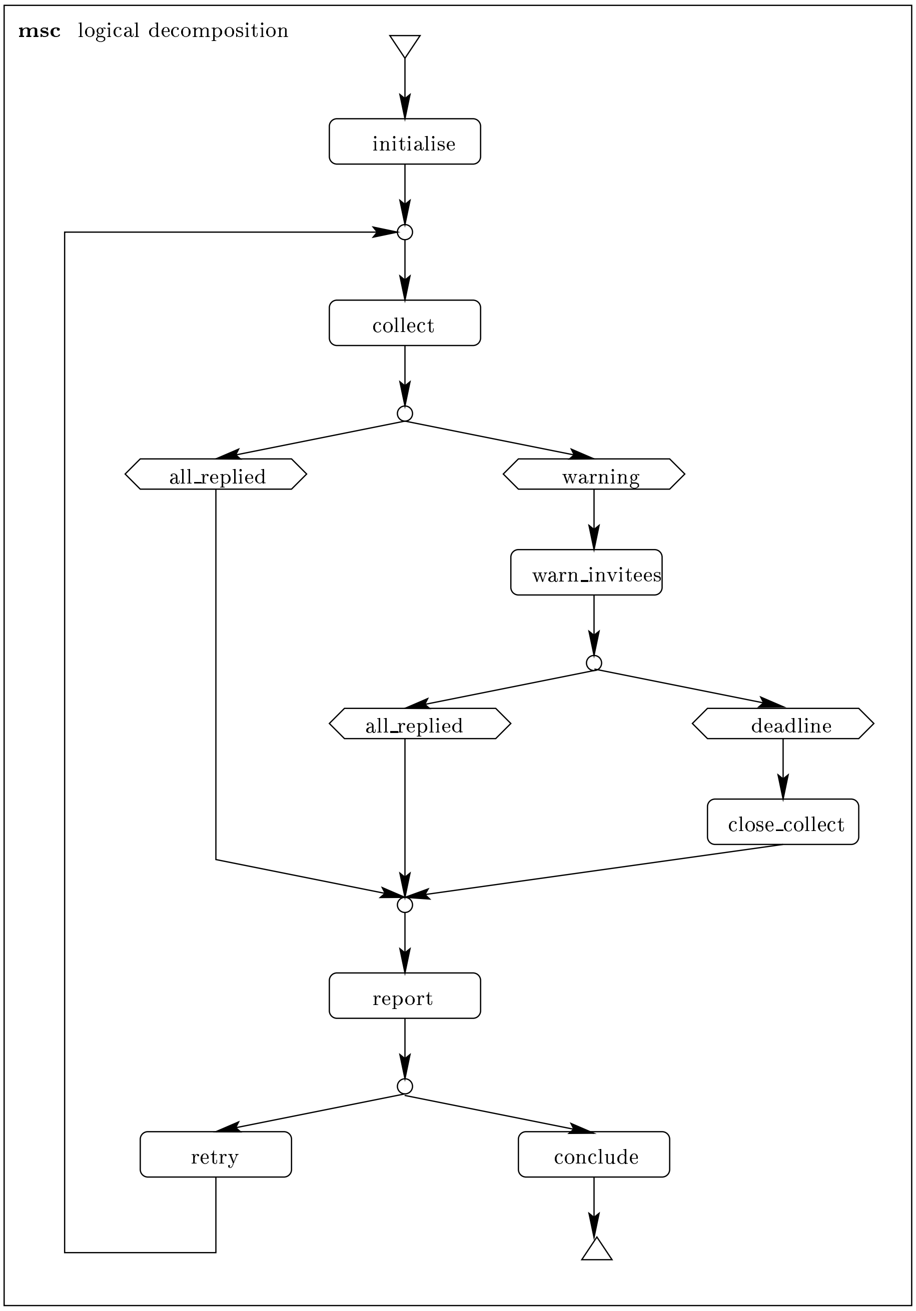

Fig. 13. The global change of state for the Meeting Scheduler.

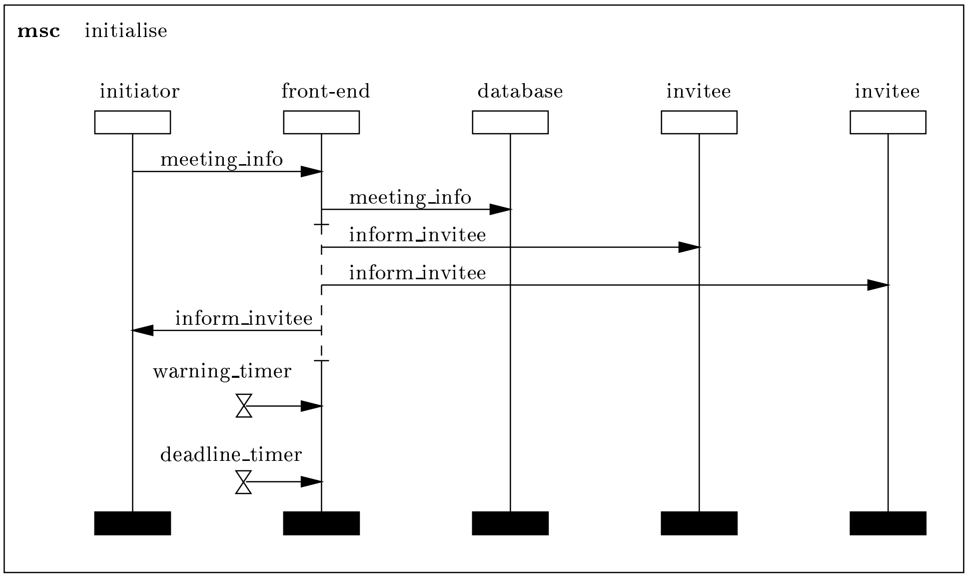

Fig. 14. The initialisation of the Meeting Scheduler.

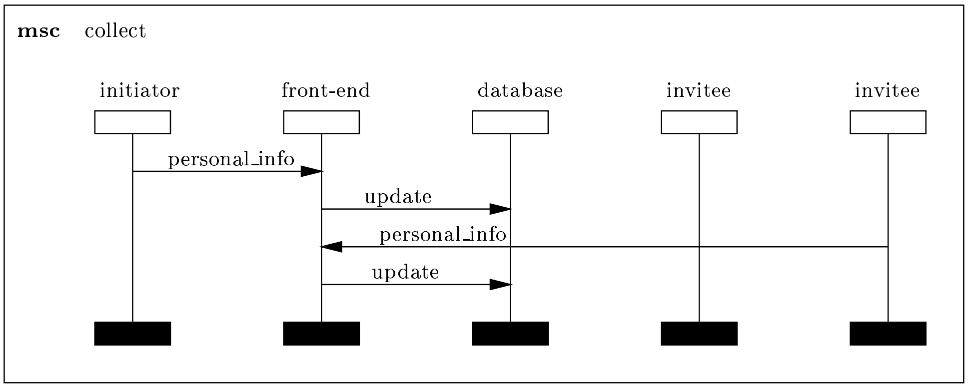

Fig. 15. Collecting of information.

the invitees inform the Meeting Scheduler of their preferred dates and times for the meeting to take place. The front-end of the system uses this information to update the database via an update message.

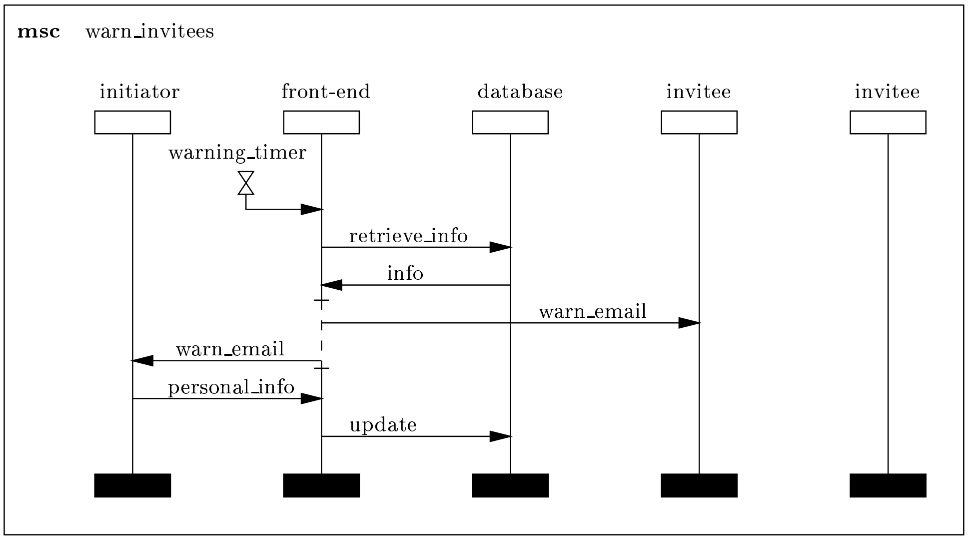

Fig. 16. Warning of invitees.

As an example of how non-functional requirements are captured in this phase, the MSC warn_collect is illustrated in Fig. 16. When a warning timeout is reported to the front-end system, the front-end system consults the database for information about who responded and who did not yet respond to the meeting announcement. Those invitees that did not yet respond, are warned (expressed by the message warn_email) by the front-end subsystem.

Validation in this phase boils down to checking whether the MSC scenarios described in this phase refine the MSC scenarios generated in the specification phase. For the Meeting Scheduler, the interactions left after abstracting from the interactions between the different subsystems of the Meeting Scheduler should also be allowed scenarios of the specification.

5. Concluding remarks

This paper presented an overview of the canonical applications of the language *Message Sequence Chart* (MSC) in the area of software engineering. These applications have been sketched independently of any particular software engineering methodology (e.g. ESA PSS 5 Software Engineering standard) or model (e.g. the incremental delivery model, the waterfall model). Alongside a more abstract framework, describing these applications, a more concrete example, in the form of a case study has been discussed. This allowed for relating the more practical aspects to the abstract framework.

The MSC language constructs used and indicated in this paper are but a list of the more common constructs. In particular, the case study is of a simple nature; hardly any need for structuring mechanisms exists nor does data play a role in this case study. Yet, for illustrating some of the canonical applications of MSC an easy to understand example is vital. In general, the full set of language constructs does allow for dealing with substantially more complex applications than the ones sketched in this paper. In dealing with such applications, it is recommended (and often necessary) to choose those language constructs necessary for describing exactly what is relevant for the particular application.

Some aspects of the applications mentioned in the preceding sections are quite orthogonal with respect to others, e.g. in simulating an executable specification absolute time stamps are commonly used, whereas in writing requirements for a system, relative time is often employed. This illustrates the broad spectrum of applications for MSC we have sketched in this paper. Given this broad spectrum, it is common sense to make a selection of which purposes are best served using MSC (e.g. using MSC only during the requirements engineering phase and validation of these requirements).

As mentioned before, the formal semantics of the language MSC allows for performing validation and verification. These activities have been described without going into too much detail. Wherever possible, references to illustrative works on this area have been added.

Acknowledgements

We like to thank Victor Bos and André Engels for their efforts and we thank Lies Kwikkers and Jan Roelof de Pijper for their work on the case study.

References

1. ITU-T. *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 2000.
2. O. Færgemand and R. Reed, editors. *SDL'91 - Evolving Methods*. North-Holland, 1991.
3. O. Færgemand and A. Sarma, editors. *SDL'93 - Using Objects*. North-Holland, 1993.
4. R. Bræk and A. Sarma, editors. *SDL'95 - with MSC in CASE*. North-Holland, 1995.
5. A. Cavalli and A. Sarma, editors. *SDL'97: Time for Testing - SDL, MSC and Trends*. North-Holland, 1997.
6. R. Dssouli, G. von Bochmann, and Y. Lahav, editors. *SDL'99: Proceedings of the Ninth SDL Forum*. North-Holland, 1999.
7. E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996. Special issue on SDL and MSC, guest editor Ø. Haugen.
8. Ø Haugen. Msc-2000 interaction diagrams for the new millenium. To appear in Computer Networks and ISDN Systems, 2000, 2000.
9. Telelogic AB. *SDT 3.1 Reference Manual*. Malmö, Sweden, 1996.
10. Verilog. *ObjectGEODE Toolset Documentation*, 1996.

11. ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, June 1994.
12. R. Saracco, R. Reed, and J.R.W. Smith. *Telecommunications Systems Engeneering Using SDL*. North-Holland, Amsterdam, 1989.
13. F. Belina, D. Hogrefe, and A. Sarma. *SDL - with applications from protocol specification*. The BCS Practitioners Series. Prentice-Hall International, London/Englewood Cliffs, 1991.
14. R. Bræk and Haugen Ø. *Engineering Real-time Systems with an Object-oriented Methodology based on SDL*. Prentice-Hall International, London, 1993.
15. A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92*. Elsevier Science Publishers B.V., Amsterdam, 1994.
16. ITU-T. *Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts*. ITU-T, Geneva, 1998.
17. S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.
18. S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996. Special issue on SDL and MSC, guest editor Ø. Haugen.
19. S. Mauw and M.A. Reniers. High-level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 291–306, Evry, France, 23-26 September 1997. Amsterdam, North-Holland.
20. M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, June 1999.
21. W.W. Royce. Managing the development of large software systems. In *Proceedings of the IEEE WESCON*, 1970.
22. R.T. Yeh. An alternate paradigm for software evolution. In P.A. In Ng and R.T. Yeh, editors, *Modern Software Engineering: Foundations and Perspectives*, New York, NY, 1990. Van Nostrand Reinhold.
23. B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
24. Ministry of the Futenon, Ottobrun, Germany. *Software life-cycle process model (V-model)*, 1992.
25. C. Gindre and F. Sada. A development in Eiffel: Design and implementation of a network simulator. *Journal of Object-Oriented Programming*, 2(2):27–33, May 1989.
26. M. Andersson and J. Bergstrand. Formalizing Use Cases with Message Sequence Charts. Master's thesis, Lund Institute of Technology, 1995.
27. E. Rudolph, J. Grabowski, and P. Graubmann. Towards a harmonization of UML-sequence diagrams and MSC. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99: Proceedings of the Ninth SDL Forum*. North-Holland, 1999.
28. G. Robert, F. Khendek, and P. Grogono. Deriving an SDL specification with a given architecture from a set of MSCs. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, pages 197–212, Evry, France, 1997. Elsevier Science Publishers B.V.
29. S. Somé and R. Dssouli. Using a logical approach for specification generation from message sequence charts. Technical Report Publication départementale 1064, Département IRO, Université de Montréal, April 1997.
30. L.M.G. Feijs. Generating FSMs from Interworkings. *Distributed Computing*, 12(1):31–40, 1999.
31. I. Krüger, R. Grosu, P. Scholz, and M. Broy. *From MSCs to Statecharts*. Kluwer Bedrijfswetenschappen B.V., 1999.

32. S. Leue, L. Mehrmann, and M. Rezai. Synthesizing room models from Message Sequence Chart specifications. Technical Report Technical Report 98-06, Department of Electrical and Computer Engineering, University of Waterloo, April 1998.
33. D. Toggweiler, J. Grabowski, and D. Hogrefe. Partial order simulation of SDL specifications. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 293–306, Oslo, 1995. Amsterdam, North-Holland.
34. G.J. Holzmann. The model chacker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
35. A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for Message Sequence Charts. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, Proceedings of FORTE X and PSTV XVII '97, pages 75–90, Osaka, Japan, November 1997. Chapman & Hall.
36. B. Takacs. Use of SDL in an Object Oriented Design Process during the development of a prototype switching system. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 79–88, Darmstadt, 1993. Amsterdam, North-Holland.
37. Ø Haugen, R. Bræk, and G. Melby. The SISU project. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 479–489, Darmstadt, 1993. Amsterdam, North-Holland.
38. Ø. Haugen. Using MSC-92 effectively. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 37–49, Oslo, 1995. Amsterdam, North-Holland.
39. G. Amsjø and A. Nyeng. SDL-based software development in Siemens A/S – experience of introducing rigorous use of SDL and MSC. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 339–348, Oslo, 1995. Amsterdam, North-Holland.
40. L.M.G. Feijs, F.A.C. Meijs, J.R. Moonen, and J.J. van Wamel. Conformance testing of a multimedia chip using PHACT. In A. Petrenko and N. Yevtushenko, editors, *Testing of Communicating Systems*, pages 193–210, 1998.
41. J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 253–265, Darmstadt, 1993. Amsterdam, North-Holland.
42. J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, Universität Bern, 1994.
43. R. Nahm. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts*. PhD thesis, Universität Bern, 1994.
44. J. Grabowski, R. Scheuer, Z.R. Dai, and D. Hogrefe. Applying SaMsTaG to the B-ISDN protocol SSCOP. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, IFIP TC6 Tenth International Workshop on Testing of Communicating Systems, pages 397–415, Cheju Island, Korea, September 1997. Chapman & Hall.
45. ISO. *TTCN: ISO/IEC JTC 1/SC 21: Information Technolgy - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation*, volume ISO 9646-3. ISO/IEC, 1991.
46. E. Algaba, M. Monedero, E. Pérez, and O. Valcárel. HARPO: Testing tools development. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, IFIP TC6 Tenth International Workshop on Testing of Communicating Systems, pages 318–323, Cheju Island, Korea, September 1997. Chapman & Hall.
47. E. Pérez, E. Algaba, and M. Monedero. A pragmatic approach to test generation. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*,

IFIP TC6 Tenth International Workshop on Testing of Communicating Systems, pages 365–380, Cheju Island, Korea, September 1997. Chapman & Hall.
48. J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger. Test case specification based on MSCs and ASN.1. In R. Bræk and A. Sarma, editors, *SDL'95 - with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 307–322, Oslo, 1995. Amsterdam, North-Holland.
49. L.M.G. Feijs and M. Jumelet. A rigorous and practical approach to service testing. In B. Baumgarten, H. Burkhardt, and A. Giessler, editors, *Testing of Communicating Systems*, IFIP TC6 Nineth International Workshop on Testing of Communicating Systems, pages 175–190. Chapman & Hall, 1996.
50. A. Cavalli, B. Lee, and T. Macavei. Test generation for the SSCOP-ATM networks protocol. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 277–288, Evry, 1997. Amsterdam, North-Holland.
51. A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer-Verlag, 1997.
52. S. Mauw and M.A. Reniers. A process algebra for interworkings. Technical Report CSR 00/03, Eindhoven University of Technology, Department of Computing Science, 2000. To appear as a chapter in Handbook of Process Algebra, editors A. Ponse and S. Smolka, Elsevier Science B.V., 2000.
53. R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, S. Sikkel, J. Trevor, and G. Woetzel. Basic support for cooperative work on the world wide web. *International Journal of Human-Computer Studies*, 46(6):827–846, 1997.