

# Synthesis of Concurrent Haskell programs from Message Sequence Charts

Volker Stolz

RWTH Aachen, 52056 Aachen, Germany

**Abstract.** In this article we show how to generate executable Concurrent Haskell programs from message sequence charts (MSCs). MSCs capture the communication between processes in a concurrent system and allow for a concise graphical and textual specification. Their modularity and extensibility can be reflected by Haskell's module concept. By compiling the generated code, several of the static properties of MSCs are verified by the Haskell compiler.

## 1 Introduction

Concurrent or distributed systems are usually specified by a formal method before they are implemented. Ideally, verification techniques which determine possible deadlocks or other properties the system should have are employed to guarantee that the systems shows the desired behaviour. The implementation however is mostly done in a separate step which might introduce several kinds of bugs: On the one hand, the implementation might simply not implement what the specification specified, or, on the other hand it might introduce new bugs which do not exist on the specification level. Therefore, it is important to have a standardised way of deriving the implementation from the specification.

Message Sequence Charts (MSCs) [13] are one way of specifying the behaviour of a concurrent system. They consist of a set of processes (called *instances*) and events ordered on the time axis of each instance. They are mostly used for documentation or test-case specification. Usually in these cases, an MSC does not show the entire system but just a part of an execution trace, although MSCs can include branching or iteration. [23] describes automatic synthesis from MSCs within the MESA toolset. In the same spirit as Live Sequence Charts [28], we adapt MSCs as an input to code generation.

In this article we show how to generate executable Concurrent Haskell programs or skeletons from MSCs. Static verification of properties which have to hold for an MSC to be valid is done by the Haskell compiler through type checking. This includes message passing, concatenation (executing one MSC after another) or composition (communication between instances specified in separate MSCs).

In the next section, we will give a short overview of Concurrent Haskell. Section 3 introduces the salient features of MSCs. In Section 4 we show how to generate code from a specification and using the type checker to assure validity of the MSC. Section 5 concludes.

## 2 Concurrent Haskell

For concurrent programming in the lazy functional programming language Haskell [17], Jones, Gordon and Finne proposed Concurrent Haskell [16] which integrates concurrent lightweight threads.

Concurrent Haskell offers a variety of concepts for communication between different threads, all based on mutable variables (`MVar`). Mutable variables are embedded in the IO monad [15]. This is necessary because of the nondeterminism of the underlying interleaving semantics. Different schedules may lead to different communication taking place and therefore to different results. In Concurrent Haskell, inter-thread communication is a side-effect. Based on `MVars` a couple of higher-level objects are built, most notably a data structure for unbounded FIFO-queues called `Channel`.

Channels are useful for asynchronous communication between several threads as the sender does not have to wait for a `writeChan` to succeed: writing to a full `MVar` would block. On the receiving end of the channel (which can have several readers), messages will be retrieved sequentially in the order they have been written into it. Reading will block until another item arrives if the channel is empty. The basic concurrency primitives we will be using are:

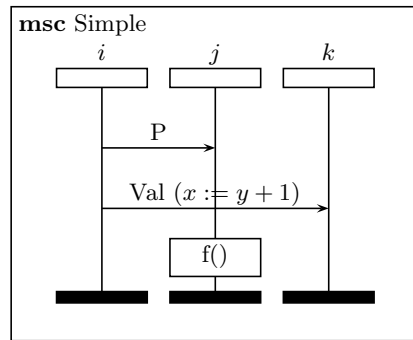
```
- newChan    :: IO (Chan a)           - creates a new channel
- readChan   :: Chan a -> IO a        - read from a channel
- writeChan  :: Chan a -> a -> IO ()  - write an item to a channel
- forkIO     :: IO a -> IO ThreadId  - start a new concurrent process
```

## 3 Message Sequence Charts

Message Sequence Charts describe the communication behaviour of a concurrent system. An MSC consists of a set of *instances* and the *instance events* these instances execute (ordered relatively in time). Instance events may involve one or more instances: A specific event might be some action which might print a result to the screen, send a message (which involves a sender and a receiver) or a condition which spans a set of instances and has to be satisfied to continue. In the following we describe the different features relevant for our implementation. We do not treat timers, message overtaking, lost/found messages and coregions, where the general ordering has been rescinded.

The basic building blocks for MSCs are messages. A message is sent from the sending process to the recipient. The medium is implicitly considered buffered. Together with local actions (informal text with no special meaning, formal data statements will be discussed later), these simple types of MSCs are often referred to as basic MSCs.

Figure 1 shows a basic MSC with two messages (one containing dynamic data) and one action, in graphical and in phrase representation. The meaning of basic MSCs can be given in process algebra semantics [11]. As all instance events are ordered on the time axis, we can derive a partial ordering of events from the MSC: Output events have to occur before their corresponding input events. Events on the same instance are executed sequentially (cf. Figure 2).



```

msc Simple;
  instance i;
    out P to j;
    out Val (x:=y+1) to k;
  endinstance;
  instance j;
    in P from i;
    action "f()";
  endinstance;
  instance k;
    in Val (x:=y+1) from i;
  endinstance;
endmsc;

```

Fig. 1. Basic MSC

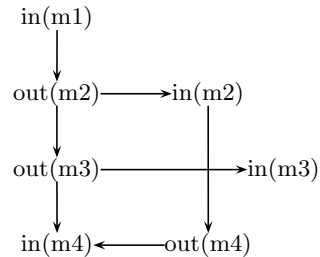
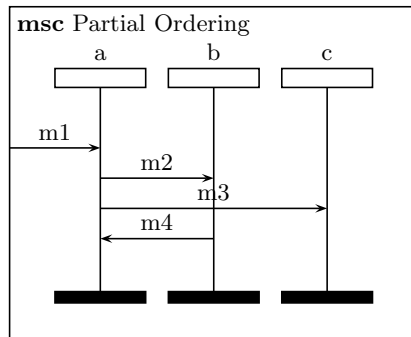


Fig. 2. Partial ordering of events

Apart from a message constructor (“*Val*” or “*P*” in Figure 1), messages can also contain additional data. This can either be static data or data obtained dynamically from the current environment. Through *bindings* (which bear a striking resemblance to Haskell’s variable assignment through pattern matching) it is possible to bind a value which has been received through a message in the environment of the receiving instance, overwriting a pre-existing binding. Syntactically a binding consists of a pattern part (variable “*x*” in Figure 1) and an expression part (“*y + 1*”), delimited by “:=” to resemble the assignment operator from programming languages. Refer to [6] for an in-depth discussion of dynamic data. Variables are typed over basic data types: Boolean, natural numbers (including a special value “*inf*” for infinity) and time. They can be easily extended by a user-defined data language, e.g. through TTCN-2 [12].

### 3.1 Conditions

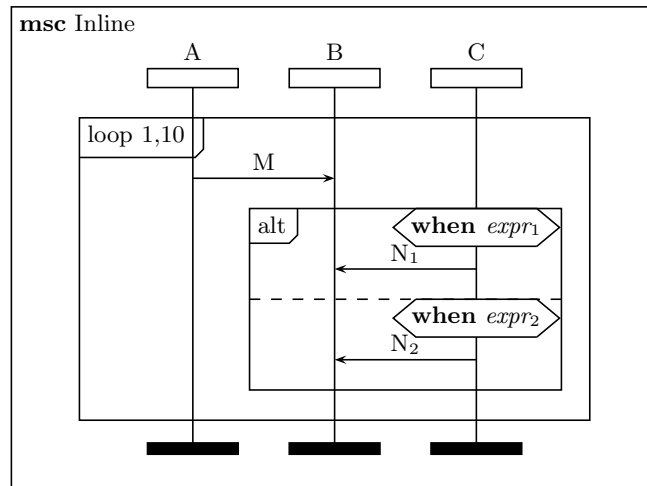
MSCs provide a way of specifying conditional execution. A *guarding condition* over one or more instances limits execution to those instances on which the con-

dition (eventually depending on dynamic values) evaluates to True. The keyword **otherwise** indicates that this guard always evaluates to True if all other guards evaluate to False. Furthermore, a *setting condition* may be used for documentary purposes or to specify which atomic condition holds in the current instance. Frequently, a setting condition simply describes the current state without any additional semantics. Such conditions at the top or the bottom of an MSC spanning all instances are also used to clarify which charts can be concatenated.

### 3.2 Inline Expressions

For more complex scenarios, *inline expressions* can be used to compose MSCs. We treat the following types of expressions:

- alternatives (**alt**)
- parallel composition (**par**)
- iteration (**loop**)



**Fig. 3.** Inline expressions

Each sub-chart of such an inline expression may be preceded by a guarding condition. These guards e.g. allow the alternative composition (“**alt**”) to be used as **if-then-else** or **switch** blocks. If more than one alternative is possible, a candidate is chosen nondeterministically. Presently we only consider alternatives and loops. Figure 3 shows some of the inline expressions in an example, the dotted line separates different alternatives.

Clearly an inline expression specifying an alternative of either “send A from  $p_1$  to  $p_2$ ” or “send B from  $p_2$  to  $p_1$ ” poses a problem for a possible concurrent implementation. When the instances reach the branch-point, they have to “agree”

on which branch of the alternative to take. Although we do not intend to prohibit this (see the discussion of non-local choice below), a system description should not contain nondeterminism. All alternatives should be guarded by mutual exclusive conditions. We discuss this phenomenon with a possible implementation in the section on code generation below.

### 3.3 Instance Creation & Termination

As MSCs mostly describe a particular part of a system, explicit instance creation and termination are rarely used. Figure 4 depicts the imagery used in the graphical representation and the corresponding keywords for the phrase representation. Instance *parent* creates a new instance called *child* with an additional argument *data*. The child process then terminates after receiving the message labelled *terminate* from *parent*.

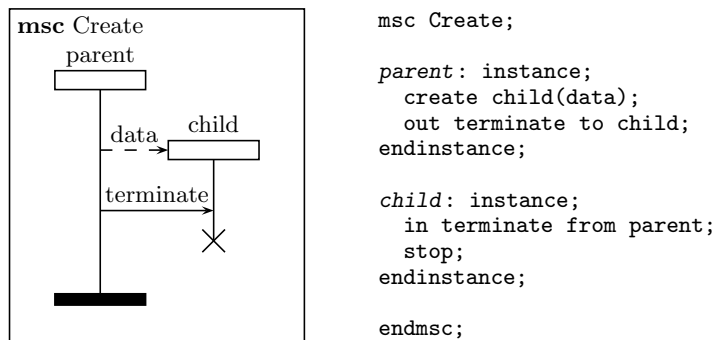


Fig. 4. Instance creation and termination

### 3.4 References and Gates

Using references it is possible to embed one MSC into another. Any case of recursion, neither direct nor indirect, to the enclosing MSC is prohibited. For connecting MSCs through references, messages to the surrounding environment are labelled with *gates*. All gates of a referenced MSC must be connected in the surrounding MSC. Messages to gates may contain the expression part of a binding, messages from gates the binding part.

### 3.5 High-level MSCs and concatenation

Our approach does not handle so-called High-level MSCs (HMSCs). These provide only a very coarse-grained view of possible compositions of MSCs which we can model using inline expression. Furthermore, the standard explicitly does not give any meaning to HMSCs (see e.g. [8] for sample applications).

There are two different notions of concatenation of MSCs [2]: The first, called *synchronous concatenation* expects that all instances wait after executing their last event, so that all instances enter the following MSC at the same time. On *asynchronous concatenation* each instance can enter the following MSC independently. This is the natural behaviour if we simply map instances to process.

### 3.6 Valid MSCs

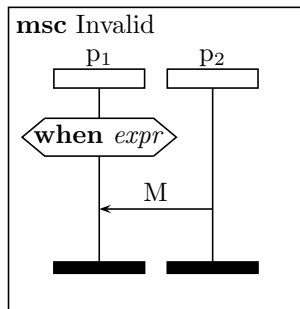


Fig. 5. Invalid MSC

the following example in Figure 5 illustrates why this is necessary: Instance  $p_2$  is not covered by condition  $expr$  and initiates a send-event to instance  $p_1$ . If we assume that both instances execute independently and concurrently in an environment where  $expr$  evaluates to False,  $p_1$  is barred from executing beyond the condition (perhaps in an alternative) while  $p_2$  would be able to proceed although both are directly related through the message event.

As MSCs may also contain dynamic data which is e.g. used for evaluation of conditions, an MSC may also be *dynamically illegal*: This is the case when for example in an alternative all guarding conditions evaluate to False.

## 4 Code Generation

Part of the success of MSCs is owed to the fact that there is a formal definition of the syntax for their textual (phrase) representation (MSC/PR) and their graphical representation (MSC/GR). Although the original grammar [13] together with later changes [14] still is not suitable for automatic parser generation, a viable subset can easily be implemented.

Given an MSC (or a set of MSCs), our system should be able to generate executable Concurrent Haskell skeletons. The static requirements should be verified by the Haskell-compiler which takes these skeletons as input, not the translation mechanism. This can be achieved by giving instances an appropriate signature to detect e.g. partial messages with only a sender or only a receiver. Furthermore, the type-checker should check that the interface to a referenced MSC matches. Types of variables must also be checked.

Concurrent Haskell's threads lend themselves naturally for implementing the instances. For ordering of the instance events, we execute them sequentially in

Haskell’s IO monad. We translate each event into an IO action. The nondeterministic interleaving-semantics of Concurrent Haskell allows any possible trace covered by the MSC semantics.

While we only consider asynchronous concatenation, synchronous concatenation can easily be achieved by introducing explicit barriers into the generated code.

#### 4.1 Communication

The message constructors in an MSC map easily to Haskell’s algebraic data types. As variables used in bindings have to have their type declared in the MSC document, we can generate the corresponding type based only on information from the MSC. This assures that one instance does not send an integer and the receiver expects a string. From the different means of communication in Concurrent Haskell, i.e. `MVars`, `Channels` or a custom made structure, we have to choose the one corresponding to the desired model. [1] examines the hierarchy resulting from different communication strategies with unbounded buffers for basic MSCs in closed systems. They distinguish five different types of communication:

- A global random access buffer shared by all instances
- A single buffer for each instance which is used both for incoming and outgoing messages
- A pair of buffers where all messages that are sent from one specific instance to another specific instance pass through
- A single unique buffer for each message
- No buffer at all, i.e. synchronous communication

Although the above classification is too fine grained, it can serve as a guideline for a possible translation. An unbounded FIFO buffer can be represented through a `Channel` in Haskell. The version using synchronous communication can be implemented using two `MVars` (one for the message and one for the explicit acknowledgement that it has been received), but is not relevant to our approach.

As typing is an issue, a global random access buffer would inconvenience us because that would mean generating a single flat algebraic data type containing all possible data types of messages sent. Refer to [26],[19] for a discussion of the problem of aggregating different message types into a single data type. The same holds for storing all incoming messages originating from various instances in a single buffer. Treating each message separately will lead to an enormous type signature for large instances if we want to type an instance over its interface. Here we have to trade off the complexity of the generated code against an additional check we have to make in the generator: The implementation for communication we will follow in this paper will be a pair of channels for each pair of communicating instances. Without a check in the generator we would not be able to detect unmatched send or receive events. But as this only applies to a single MSC (see our discussion on treatment of gates below), we can easily do this check ourselves and accumulate the message types in a single data

type for the channel. The name of the data type is derived from the involved instances (e.g. `data AtoB = ...` and `data BtoA = ...` respectively). We can thus dispense with the need for a separate buffer for each message and reduce the overhead. When translating a `loop` inline expression, we only have to pick up the message constructor once. In case of multiple alternatives, we aggregate all constructors.

Because the channel for communication has to be shared by both parties, this has some implications about initialisation and implementation of gates for referenced MSCs. Each translated MSC starts with an initialisation function where all channels for internal communication are created. For every gate we allocate a separate channel which is passed on to the initialisation of the inner MSC. The data type of the message sent via the gate is declared in the inner MSC and exported to avoid a circular module dependency. We use records to bundle the data structures used internally.

Each instance is translated into a function of type  $a_1 \rightarrow \dots \rightarrow a_n \rightarrow IO ()$  where the  $a_i$  contain the necessary channels. The functions corresponding to the outermost instances are then started concurrently from the `main` function by means of `forkIO`. In the case that execution enters a referenced MSC control is simply passed via a function call and execution resumes in the current MSC after this function returns. The compiler assures that the corresponding instance is present in the inner MSC.

The message exchange is handled by the two functions `send` and `receive` which are synonyms for `writeChan` and `readChan`. We give a simple example in Figure 6 and the generated code for the outer MSC “App” in Figure 7. Functions named `nToM` are record selectors for the channels; `mScS` are handles to the data structures from embedded MSCs.

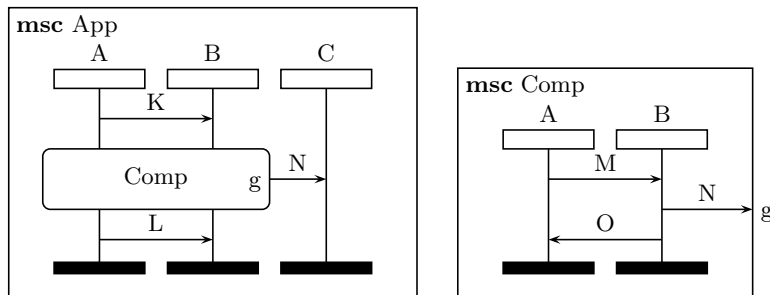


Fig. 6. Simple MSC

#### 4.2 Translating alternatives: Non-local choice and divergence

Alternatives pose the well-known problem of non-local choice: Consider the MSC in Figure 8 a). Our implementation in Concurrent Haskell allows both processes to advance independently. We have to make sure that if one process decides

```

module App where
import qualified Comp

data AtoB = K | L

data AppS = AppS {
  aTob  :: Chan AtoB,
  gToc  :: Chan Comp.OutG,
  compS :: Comp.CompS}

init :: IO AppS
init = do
  atob <- newChan
  gtoc <- newChan
  comp <- Comp.init gtoc
  return AppS
  {aTob = atob,
   gToc = gtoc,
   compS = comp}

processA aS = do
  send (aTob aS) K
  Comp.processA (compS aS)
  send (aTob aS) L

processB aS = do
  K <- receive (aTob aS)
  Comp.processB (compS aS)
  L <- receive (aTob aS)

processC aS = do
  Comp.N <- receive (gToc aS)

main :: IO ()
main = do
  aS <- App.init
  forkIO $ App.processA aS
  forkIO $ App.processB aS
  forkIO $ App.processC aS

```

Fig. 7. Generated code

on which branch to enter, the other one also follows this path. Otherwise the system would deadlock. It is obvious that there needs to be some kind of implicit communication between the two process so that they can “agree” on which branch to take. This can be implemented in a *history variable* (cf. [22]). In this spirit, we can easily add a channel into which the first process to reach such a point writes its choice. We then multiplex this channel to all involved instances so that they can read previous decisions.

On the other hand process B is determined by the actions of A in Figure 8 b). In this case, the *wait-and-see strategy* for receiving processes can dispatch to the correct alternative based on the type of the received message: Only the sending process has to make a nondeterministic choice. [8] formalises the syntactic criterion for detecting non-local choice and gives an algorithm.

Although we currently do not use this message based dispatch, we determine whether the wait-and-see strategy is applicable and generate simpler code where one instance determines the execution path and notifies the others. This proves appropriate for mutually exclusive, guarded alternatives. Refer to the database example in Figure 12 for an application.

Another problem for theoretical approaches to MSCs poses no real problem for our implementation: Divergence can occur when one process can send a message an unbounded number of times ahead of the receiving process. Once again the nondeterministic interleaving semantics of Concurrent Haskell and the unbounded nature of channels (modulo total available memory) correspond nicely to the MSC semantics: Divergence is neither prohibited nor does it necessarily affect the running system. Care has to be taken when selecting a Haskell system to run the generated source code: Only a preemptive runtime-system can assure

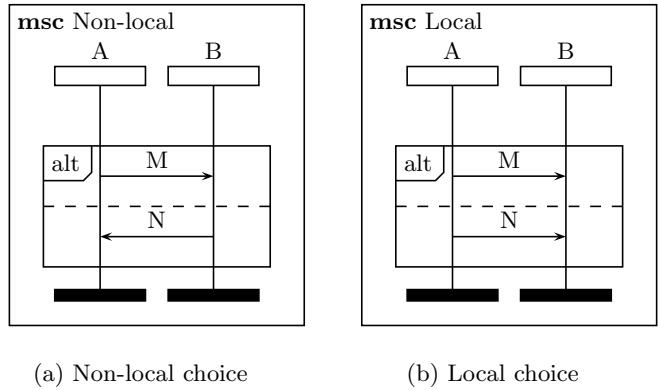


Fig. 8. Branching

that all ready processes get a chance to run eventually. In a cooperative environment such as provided by Hugs (The Haskell User's Gofer System), divergence can indeed mean that the systems enters a live-lock and eventually exhausts the memory.

### 4.3 Inline expressions and gates

The translation of MSCs where gates are exported from inline expressions poses an interesting question for type checking. If we consider the simple MSC in Figure 9, we can easily see that this loop can be executed an arbitrary number of times (the `loop` construct will often depend on dynamic values). Of course the receiver has to be aware of this loop, so we enforce this through the type by creating a new type of channel. Note that there is still the possibility of a runtime error as both instances are not inside the same loop-expression and may execute the loop a different number of times.

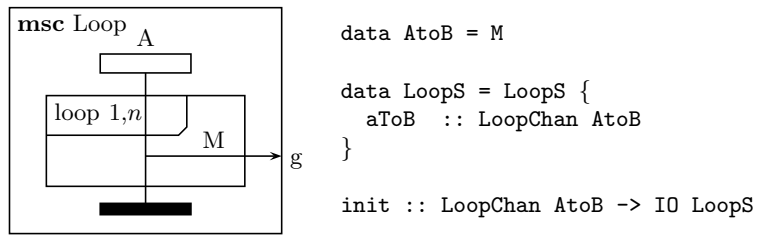


Fig. 9. Typing loops

We use a similar technique for alternatives. In this case we can resort to Haskell's `Either` data type which is defined as `data Either a b = Left a |`

Right b (example shown in Figure 10). This allows us to return the data type captured in each branch. As Z.120 allows more than two alternatives, we have to create types for more alternatives with the generated code. A choice channel is also passed through.

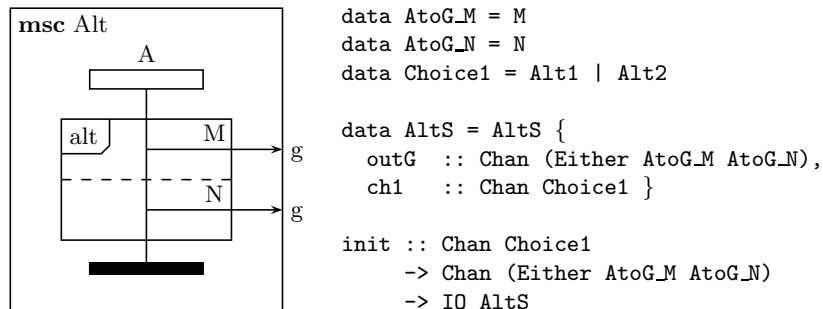


Fig. 10. Typing alternatives

#### 4.4 Extensions of MSC skeletons and dynamic data

While the discussed technique above indeed generates executable typed skeletons from MSCs, we need a way of plugging in code which is unrelated to the `send` and `receive` statements, that is where the actual application does its work apart from the communication. MSC actions provide a convenient hook for this. An action can be interpreted as the name of a fully qualified Haskell function. As in referenced MSCs, we can simply invoke this action and continue executing instance events once it returns. An action statement can also contain a binding where like in a message the expression can contain a function call to the data language. The return value is bound to the variable.

Haskell assures that these actions do not interfere with the actual communication: If the channels used for communication are not passed as an argument to the action function, there is no way for the function to get hold of them. These functions currently must have a type of `IO a` (Haskell silently drops unused results in monadic actions if they are not the last statement in a `do`-block). If the programmer wants to keep/modify the state of the system between different actions on the same instance, there needs to be a way of initialising the state and passing it on. The actions could even manipulate state variables invisible to the specification, so he needs to have a way of keeping his local state without explicitly manipulating it in the specification.

Haskell's standard library `Control.Monad.State` provides exactly the feature we need: It encapsulates a state and provides wrappers for accessing the content. By lifting our current scheme from type `IO a` and `IO ()` for the communication statements respectively to a new monad `MIO`, we can do this for the programmer transparently. We assume that user-defined types and actions are

in a separate module (module `MscUser` in the following `DBQuery`-example would contain the database or database connection). The MSC recommendation only allows a single type for a variable in its lifetime, so once again this mixes well with Haskell's type concept. Figure 11 shows a possible interaction of a client with a database server, Figure 12 shows the pertinent part of the corresponding generated code.

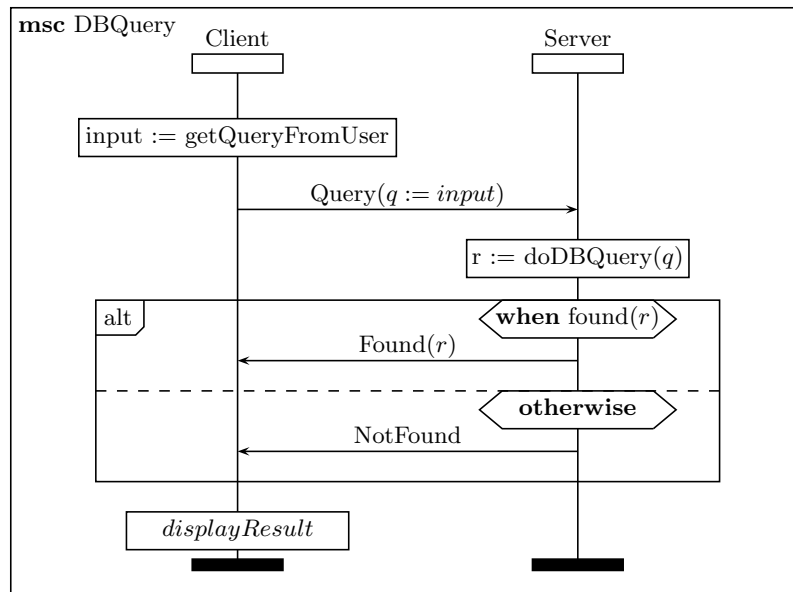


Fig. 11. MSC for database query

#### 4.5 Nondeterminism and failure

Of course it is possible for the execution to take a path which will lead to a runtime error, e.g. when all alternatives are blocked by guards. It should be possible to wind back the execution to the point where the latest branch-decision was taken: Using Haskell's exception-handling, we could capture the failed computation and nondeterministically take a new decision. Naturally any side-effects which took place meanwhile, e.g. in action blocks, cannot be reverted. It is also not feasible to notify the communication partner of this exception. But if we only permit such an exception for the first message a process receives in a nondeterministic alternative, this could be used to implement the wait-and-see-strategy mentioned before. Notice that this would also mean putting the message which triggered the error once more into the input queue. Writing to the head of a FIFO channel is supported by Haskell's `unGetChan :: Chan a -> a -> IO ()` function. However, uncontrolled nesting of exception handlers could lead to memory exhaustion.

```

data CtoS = Query Int
data StoC_NotFound = NotFound
data StoC_Found = Found Int
data StoC = Either StoC_Found StoC_NotFound
data Choice1 = Alt1 | Alt2

data MscS = MscS {
    cToS :: Chan CtoS,
    sToC :: Chan StoC,
    ch1  :: Chan Choice1
}

data StateCS = StateCS {
    localC :: MscUser.State,
    inputC :: Int,
    rC     :: Int
}

data StateSS = StateSS {
    localS :: MscUser.State,
    qS     :: Int,
    rS     :: Int
}

getQueryFromUser :: MIO StateCS Int

displayResult :: MIO StateCS ()

doDBQuery :: Int -> MIO StateSS Int

c :: MscS -> MIO StateCS ()
c mscS = do
    input <- MscUser.getQueryFromUser
    get >>= \ st -> put st { inputC = input }
    input <- gets inputC
    send (cToS mscS) (Query input)
    c <- lift $ readChan (ch1 mscS) -- Choice!
    case c of
        Alt1 -> do
            Left (Found r) <- receive (sToC mscS)
            get >>= \ st -> put st { rC = r }
        Alt2 -> do
            Right NotFound <- receive (sToC mscS)
            get >>= \ st -> put st { rC = -1 }
    MscUser.displayResult

s :: MscS -> MIO StateSS ()
s mscS = do
    Query q <- receive (cToS mscS)
    get >>= \ st -> put st { qS = q }
    q <- gets qS
    r <- MscUser.doDBQuery q
    get >>= \ st -> put st { rS = r }
    if (found r)
    then do
        lift $ writeChan (ch1 mscS) Alt1
        r <- gets rS
        send (sToC mscS) (Left (Found r))
    else do
        lift $ writeChan (ch1 mscS) Alt2
        send (sToC mscS) (Right NotFound)

```

**Fig. 12.** Generated code with variables and state

The use of dynamic failure closely resembles Harel’s *cold conditions* in Live Sequence Charts [4] where a failed guard causes a different alternative to be tried. Such an exception can also be used to implement *hot conditions* which are used like assertions and specify conditions that must hold at this place and indicate terminal failure if violated. This could also be a point of integration with the Concurrent Haskell LTL runtime verifier [27].

#### 4.6 Limitations

The approach of using MSC for system specification is currently limited to closed systems where all instances are either present from the beginning or are explicitly create by another process. It is not possible to specify an open distributed system where new instances are dynamically created, e.g. a database server talking to several clients on a network. In a distributed setting, also timers and lost messages should be supported.

From the database example it is clear that a better way for doing message based dispatch is needed. Exceptions on a failed pattern matching or an unsatisfied condition could solve this. This would generate clearer code and would be able to correctly treat more MSCs. Currently there are still unhandled cases where the system would deadlock or fail. Also not all possible MSCs generate executable programs, especially when gates “protruding” from inline expressions are used. Compiler error messages from invalid input are not generally helpful in finding errors in MSCs. Although for some errors resulting from the static requirements it should be possible to translate them to a (natural language) description of the error.

## 5 Related Work and Conclusion

Message sequence charts are mostly used for either documenting or specifying sample runs of a concurrent system and aid in test case generation [25], one such system even uses Standard ML [20,21]. Although MSCs have also been considered during the design phase, deriving an implementation is usually deferred: [10] statecharts are generated which can be used in the STATEMATE framework [7], the commercial tool RHAPSODY is an even more complex CASE tool [5]. [23] synthesises Real-Time Object-Oriented Modelling (ROOM) models which can be executed inside the MESA toolset [9]. Live sequence charts [28] address some further perceived deficiencies of MSCs. Thorough verification of MSCs is of specific interest, e.g. in [2] through model checking, or by automatically verifying properties with SPIN [24]. The latter approach can also be used to execute the specified runs. [3] gives an execution semantics for the current MSC recommendation which also includes dynamic data<sup>1</sup>.

We have presented a framework for generating executable Concurrent Haskell programs from an MSC specification. From basic MSCs with static messages

---

<sup>1</sup> We have compiled a comprehensive list of tools which work with MSCs at <http://www-i2.informatik.rwth-aachen.de/Research/AG/MCS/MSC/>.

through complex MSCs with references and inline expressions we outlined the necessary design decisions to make so that we can provide flexibility to the programmer and use Haskell's type checking to verify the static requirements for valid MSCs. The system can generate code for a large set of MSCs. Concurrent Haskell's interleaving semantics for IO actions covers the nondeterministic execution of instance events based only on their causal ordering. This approach permits a formal basis for the specification of concurrent systems in Haskell and allows to reuse tools for Concurrent Haskell such as the Concurrent Haskell Debugger or the runtime verifier. Currently we are also working on a related approach of generating code for JAVA which already is able to execute a single MSC without references.

## References

1. 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, 1997. Chapman & Hall.
2. R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proc. 10th Intl. Conf. on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129, Eindhoven, Netherlands, August 1999. Springer.
3. B.Jonsson and G.Padilla. An Execution Semantics for MSC-2000. In *Proc. of the 10th International SDL Forum*, volume 2078 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, June 2001. Springer.
4. D.Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.
5. D.Harel and E.Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
6. A. Engels. Design Decisions on Data and Guards in MSC2000. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM2000. 2nd Workshop on SDL and MSC*, Col de Porte, Grenoble, June 2000.
7. D.Harel et al. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.
8. H.Ben-Abdallah and S.Leue. Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts. In E.Brinksma, editor, *Proc. of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 259–274, Enschede, The Netherlands, April 1997. Springer.
9. H.Ben-Abdallah and S.Leue. MESA: Support for Scenario-Based Design of Concurrent Systems. In B.Steffen, editor, *Proc. of 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, Lisbon, Portugal, March/April 1998. Springer.
10. I.Krüger, R.Grosu, P.Scholz, and M.Broy. From MSCs to Statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

11. ITU-T. *ITU-T Recommendation Z.120 Annex B: Algebraic Semantics of Message Sequence Charts*. International Telecommunications Union – Telecommunications Standards Sector, 1995.
12. ITU-T. *ITU-T Recommendation X.292: TTCN-2 Standard, Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. International Telecommunications Union – Telecommunications Standards Sector, 1997.
13. ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. International Telecommunications Union – Telecommunications Standards Sector, 1999.
14. ITU-T. *ITU-T Recommendation Z.120 (1999) Corrigendum 1*. International Telecommunications Union – Telecommunications Standards Sector, 2001.
15. S.Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. <http://research.microsoft.com/~simonpj/#marktoberdorf>, January 2001.
16. S.Peyton Jones, A.Gordon, and S.Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23<sup>rd</sup> ACM Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 1996.
17. S.Peyton Jones et al. Haskell 98 Report. Technical report, <http://www.haskell.org/>, 1998.
18. M.A.Reniers. Static semantics of Message Sequence Charts. In *Proc. of the 7th SDL Forum*, Oslo, 1995. Elsevier Science Publishers B.V.
19. M.Wallace and C.Runciman. Type-checked message-passing between functional processes. In Hammond, Turner, and Sansom, editors, *Proceedings of the Glasgow Functional Programming Group Workshop*, Series of Workshops in Computing, pages 245–254, Ayr, Scotland, Sept 1994. Springer.
20. P.Baker, C.Jervis, and D.J.King. An Industrial use of FP: A Tool for Generating Test Scripts from System Specifications. In G.Michaelson, P.Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming*. Intellect, 2000.
21. P.Baker, P.Bristow, C.Jervis, D.J.King, and B.Mitchell. Automatic Generation of Conformance Tests from Message Sequence Charts. In E.Sherrat, editor, *Telecommunications and beyond: The Broader Applicability of SDL and MSC: Third International Workshop, SAM 2002*, volume 2599 of *Lecture Notes in Computer Science*, pages 170–198, Aberystwyth, UK, 2003. Springer.
22. P.B.Ladkin and S.Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
23. S.Leue, L.Mehrmann, and M.Rezai. Synthesizing ROOM Models from Message Sequence Chart Specifications. In *13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, October 1998.
24. S.Leue and P.B.Ladkin. Implementing and verifying MSC specifications using Promela/XSpin. In *Proc. of of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System*.
25. S.Mauw, M.A.Reniers, and T.A.C.Willems. Message Sequence Charts in the software engineering process. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*. World Scientific, 2000.
26. V.Stolz. Robuste verteilte Programmierung in Haskell. Master’s thesis, RWTH Aachen, 2001.
27. V.Stolz and F.Huch. Runtime Verification of Concurrent Haskell. In Germán Vidal, editor, *Draft Proc. of the 12th Int. Workshop on Functional and (Constraint) Logic Programming*, Valencia, Spain, 2003.
28. W.Damm and D.Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.