Synthesizing ROOM Models from Message Sequence Chart Specifications

Stefan Leue, **Lars Mehrmann**, and **Mohammad Rezai**
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

Technical Report 98-06

sleue|mehrman|mrezai@swen.uwaterloo.ca

April 1998

Abstract

Message Sequence Chart (MSC) specifications have found their way into many software engineering methodologies and CASE tools, in particular in the area of telecommunications and concurrent real-time systems. MSC Specifications often represent early life-cycle requirements and high-level design specifications. We are considering iterating and branching MSC specifications according to ITU-T Recommendation Z.120. We show how these specifications can be analyzed with respect to their software architectural content, including structure and behavior. We present algorithms for the automated synthesis of Real-Time Object-Oriented Modeling (ROOM) models from MSC specifications and discuss their implementation in the Mesa toolset. The automation of the synthesis contributes to making the transition from high-level, message exchange-oriented views to the level of a full life-cycle architecture description more efficient and reliable. This means that we are contributing to making Z.120 MSC specifications more useful in the software engineering process.

Contents

List of Figures

1 Introduction

Message Sequence Charts (MSCs) have been adopted within several software engineering methodologies and tools, e.g., [24], [14], [25], [9], [16], [27], [8], [1], [2], and [18]. MSCs are used to document system requirements that guide the system design [27], describe test scenarios (e.g., [16, 8]), express system properties that are verified against SDL specifications [1], visualize sample behavior of a simulated system specification [27, 1, 12], capture early life-cycle requirements [2, 13], allow for derivation of state machines from overlapping scenario diagrams [18], and express legacy specifications in an intermediate representation that helps in software maintenance and reengineering [14].

Positioning of MSCs in the Software Lifecycle. The positioning of the use of MSCs in the software life cycle is still being debated in the literature. However, as [13] and [18] argue MSCs are suitable to express early life-cycle requirements and high-level object designs. In [4, 7] we argue that due to their focused expressiveness MSCs are not a full life-cycle notation. Instead, MSCs should be a front-end to full life-cycle methods and tools. Figure 1 illustrates the use of MSCs in the software lifecycle. The development of the Mesa toolset [7] is based on this model. Initially, designers will describe sets of scenarios, and compose them to form an iterating and branching model of the system behavior. This model will be analyzed for syntactic sanity as well as semantic consistency using an approach outlined in [7]. The crucial step is then to translate the high-level MSC model into an architectural model supporting the design of the implementation.

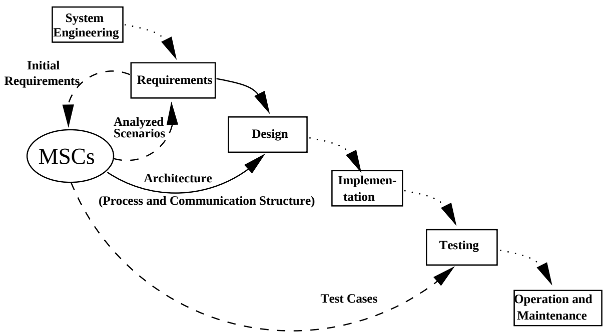


Figure 1: MSCs in the software lifecycle

Motivation for our work. As [3] point out, software architecture descriptions can have an important role in the design process as they support communication amongst members of the design team, they document early design decisions and make these accessible to early life-cycle validation, they reveal "transferable abstractions" of the system that can be re-used for the design of other, similar systems, and they can be used to generate code for later life-cycle stages. We envisage that there are two main groups that could benefit from the suggested approach: those that intend to design their system using the synthesized ROOM models by providing futher refinements, and those who are mainly interested in an executable architectural model in order to, for example, derive test drivers or validation models.

Further motivation for our work stems from the fact that the manual translation from an MSC specification into a full life-cycle modeling language like ROOM will be an error-prone and time-consuming task. To increase reliability and efficiency of the derivation of a design model from MSC specifications we will present algorithms that will automate this translation and we will describe the implementation of these algorithms as an extension of the MESA toolset [7].

Organization of Paper. In Section 2 we discuss related work as well as the methods that we are using. In Section 3 we analyze, what architectural information is contained in an MSC specification, and we define the algorithms that automatically extract the architectural information in Section 4. In Section 5 we discuss the potential use of the synthesized architectural models, and in Section 6 we illustrate the application of our algorithms to a more complex specification example.

2 Related Work

2.1 Message Sequence Chart Specifications

ITU-T Recommendation Z.120 [15] specifies two major syntactical objects as components of the MSC notation: *basic MSCs* (bMSCs) and *High-Level MSCs* (HMSCs). A bMSC essentially consists of a set of processes (called instances in Z.120) that run in parallel and exchange messages in a one-to-one, asynchronous fashion. Processes are depicted by vertical lines, message exchanges by horizontal arrows. In addition to exchanging messages, processes can individually execute internal *actions*, use *timers* to express timing constraints, create and terminate process instances. However, we will not treat this part of the Z.120 language in the course of this paper. The graph of an HMSC provides for operators to connect *basic* MSCs to describe parallel, sequential, iterating, and non-deterministic execution of basic MSCs. In addition, HMSCs can describe a system in a hierarchical fashion by referring to HMSCs within a node of an HMSC graph. We will call the collection of an HMSC graph and the referenced bMSCs an *MSC specification*. MSC specifications specify partially ordered event sequences. A number of approaches to formalizing MSCs and defining their semantics have been proposed. We will use the formalization given in [20] within the remainder of this paper.

The TOASTER example. For illustrative purposes we developed the TOASTER example MSC specification, c.f. Figure 2. The specification consists of the HMSC `TOASTER` which describes the composition of bMSCs `IDLE`, `EJECT`, `ERROR`, `START` and `TOAST`. Each of the bMSCs consists of the three processes `USER`, `CONTROL` and `HEATING`[1]. From a global perspective, the behavior of the toaster can be explained as follows. The control unit asks the user for commands and tells the heating to act accordingly. The heating, however, can report errors to the control unit that will then send a message to the user.

More specifically, in bMSC `IDLE` the `CONTROL` is asking `USER` for a command (`COMREQ`) and advises `HEATING` not to heat (`COOL`). Once the user decides to toast (`START`), the start is acknowledged (`START_ACK`) and the heating receives a command to start heating (`HOT`). In bMSC `TOAST` the beginning of the toasting process is reported to the user (`TOAST_ACK`) and a timer (`TIME`) is set

[1] Note that this toaster is not designed to ever hit the market - its somewhat contrived features will help us explain salient features of MSC specifications.

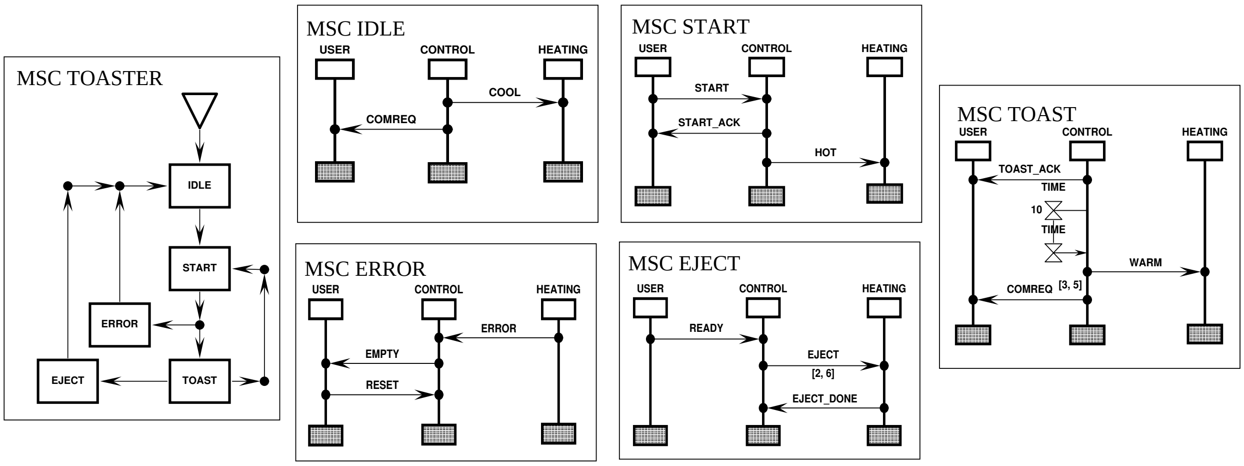


Figure 2: HMSC `TOASTER` (left) and bMSCs for TOASTER example

to ten time units to get a crusty toast. After the timer expires, the heating is switched to keep the toast warm (`WARM`) and the user is asked by message `COMREQ` to choose between another toasting period or ejecting the slice of bread. The `READY` message will cause an `EJECT` message to the heating which will respond with an `EJECT_DONE`. The toaster returns to bMSC `IDLE`. When user asks for the toasting process to start, heating will check whether there is a slice of bread in the toaster or not. If not, it will send an `ERROR` message in bMSC `ERROR` and `CONTROL` will tell the user that a the toaster is empty (`EMPTY`). Note that in the remainder of the paper we will not interpret timer events such as settings, resettings and expiry.

Model synthesis based on Sequence Diagrams. Whereas we consider iterating and branching type MSC specifications, most of the current approaches to synthesizing only consider finite scenarios or MSC specifications. [29] describe algorithms to synthesize timed automata from their own scenario description language. [26] describes the generation of test actors from basic MSCs. The model synthesis approach described by Koskimies et al. in [17, 18] is, to the best of our knowledge, the only other automated method that synthesizes state machine information from collections of more than one bMSC-like scenarios. However, the semantics of sets of scenarios that Koskimies et al. use is fundamentally different from our usage: bMSCs in their approach (called 'scenarios', reminiscent of the OMT notation) are overlapping. I.e. in a given scenario the behavior of a process is described from the beginning to the end. Each scenario corresponds to an alternative system execution and a process may at any given point in time be in different scenarios. The state machine synthesis proceeds by matching common sub-behavior in the different scenarios. Composition of bMSCs through Z.120 HMSCs as we use them here, however, assumes a mutually exclusive semantics: the control of a process lies in exactly one bMSC at any given point in time, and different bMSCs are composed in a strictly sequential fashion[2]. Hence, our synthesis algorithms greatly vary from the ones used by Koskimies et al.

[2] This interpretation of MSC composition coincides with the informal definitions in Z.120 and has also been chosen in [2, 13].

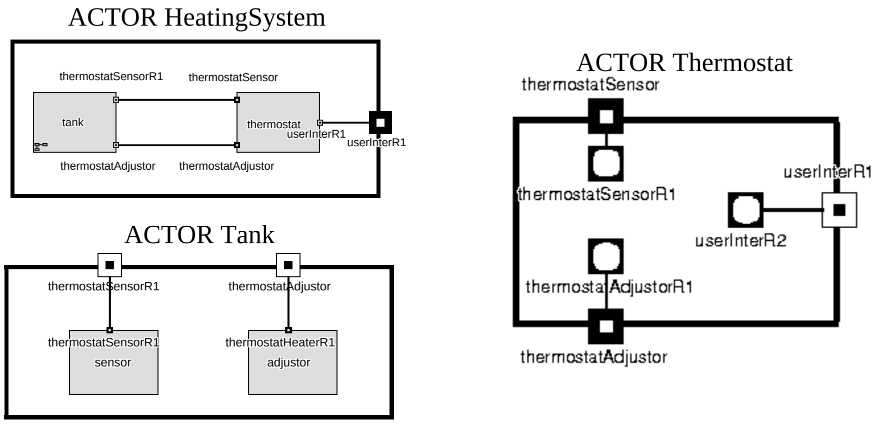


Figure 3: Structure of HeatingSystem ROOM model

2.2 Real-Time Object Oriented Modeling (ROOM)

The Real-Time Object Oriented Modeling (ROOM) [27] notation relies on two main components: the description of a system's *structure* and the dynamic *behavior* of the structural components. Structural components are defined as nested *actor* structures and their bindings. Behavior is defined using a variant of Statecharts called *ROOMcharts*. ROOM is a graphical notation for which an equivalent textual syntax, called 'linear form', has been defined in [27]. ROOM emphasizes seamless use of models from the requirements/high-level design phase down to the low-level design and testing stages, although the notation seems to have deficits when it comes to abstract requirements capture. A bMSC-like notation is used within ROOM to represent requirements on inter-object communications as well as execution traces of actual system runs. [3] have recently argued for the use of ROOM as an Architecture Description Language. The *ObjecTime Developer*[3] toolset [22] is an industrial-strength CASE tool implementing the ROOM methodology. It offers editing, simulation and code synthesis support for ROOM models. The linear form can be used within ObjecTime Developer to exchange ROOM model definitions with other CASE tools. The ROOM notation is the basis for the currently ongoing definition of the UML Real-Time notation (c.f. [28]).

The HeatingSystem example. Figures 3 and 4 show the structure and the behavior components of the ROOM model for a heating system. Space limitations do not permit us to discuss the complete ROOM notation here, for a complete presentation we refer to [27]. In Figure 3 the structural elements of the Heating System are depicted. It consists of structural entities called *actors*. The system is composed of actors `thermostat` and `tank`, where `tank` is being refined as consisting of actors `sensor` and `adjuster`. Actors in ROOM may communicate via asynchronous message exchange. Protocols (not graphically represented) are lists of *in* (receive) and *out* (send) messages. To describe a communication path between two actors, e.g. `tank` and `thermostat`, one defines a port on each of the actors (i.e. `thermostatAdjustor` on `tank` and the identically named port on `thermostat`) and connects them. This connection is called binding in ROOM. As protocols

[3]ObjecTime Developer is a trademark of ObjecTime Limited.

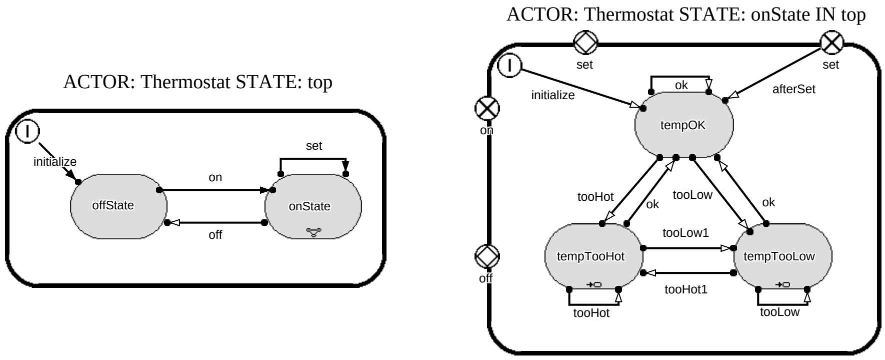


Figure 4: Behavior of HeatingSystem ROOM model

are distinguishing between in- and out messages they represent the view of just one of the communication partners. Therefore, to perform a binding two ports need to be of the same protocol type (i.e., have identical sets of messages), but one of them needs to be *conjugated* compared to the other, i.e. in and out messages need to be swapped in the conjugated protocol. The box on the right hand side of Figure 3 represents the internal structure of actor `thermostat` which has no nested structure. However, it has a behavior representing state machine (see Figure 4) attached to it. Ports can be either *relay* ports as `thermostatSensor` which are responsible to convey messages from one actor to another, or *end* ports like `thermostatSensorR1` which hand messages over to the ROOM state machines. Behavior is expressed in terms of hierarchical communicating extended finite state machines called ROOMcharts, a variant of Statecharts. The `top` state in Figure 4 is refined into states `offState` and `onState`. The `onState` diagram on the right hand side of Figure 4 consists of three different states and transitions between them. Note that while the labels on the transitions have meaningful names, e.g. "`tooHot`", these names are arbitrary and have no semantics in the model. The transition semanitcs depends on the definition of the trigger events and transition code, which is not depicted in Figure 4. The `on` and `set` points on the state boundary define the incoming transition points, and `off` and `set` define the outgoing transition points. When incoming and outgoing transition points are not connected to a particular state, this means re-entry to, or exit from, the last state in which this state machine was. For a more complete description of the ROOM notation we refer to [27].

2.3 Mesa

To support the use of Z.120 MSC specifications we have developed the Mesa toolset [7]. It allows for editing and storing Z.120 compliant MSC specifications. Furthermore, Mesa implements algorithms analyzing MSC specifications for syntactic anomalies like *non-local choice* and *process divergence* [5] and for timing inconsistency [6]. Applying the syntactic sanity checks that we defined in [5] we find that the `TOASTER` MSC specification features *non-local choice*. After going through bMSC `START` either `CONTROL` starts the toasting process or `HEATING` sends the `ERROR` signal. Due to the asynchronous nature of the processes that are involved there is the possibility that `HEATING` and and `CONTROL` will branch off into different directions which leads to a counterintuitive interpretation of the MSC specification as well as to deadlocks. We will have to pay special attention to this when we synthesize executable architectural models from the specification.

MESA is intended as a front-end to more comprehensive design notations and tools, e.g. ROOM and ObjecTime Developer, Hence, in the following Sections we will present algorithms that allow the architectural content of an MSC specification to be extracted and translated into a ROOM model, and we will illustrate the implementation of these algorithms within the MESA toolset.

3 Architectural Content of MSC Specifications

We agree with [3] that there is "*no single accepted definition of the term [software architecture]*", largely due to the fact that software architecture is an emerging discipline. We are nevertheless interested in determining what the architectural content of an MSC specification is, based on an interpretation of the notions of software architecture as discussed in [3]. As [3] argue, ROOM is suitable as an architecture description language (ADL). We discuss in this Section how the architectural content of MSC specifications can be represented in ROOM as a suitable ADL. Section 4 will present the ROOM synthesis algorithms in more detail.

3.1 Architectural Features in MSC Specifications

In following [3], we understand a software architecture to be the structure of the system, comprising the software components, externally visible properties of these components and the relationship amongst them. Furthermore, an architecture is an *abstraction* of the real system. Now, what are the architectural features in MSC specifications? Out of the list of features that [3] offers we identify the following as evident in MSC specifications:

1. *Process or coordination structure*: in an MSC specification this is given through the processes that MSC specification identifies in its bMSCs. Note that our process structure is static and that we do not consider MSC specifications with dynamic process creation or deletion. To some extent process structure also represents *conceptual* or *logical structure*, as defined in [3].

2. *Physical structure*: this aspect relates to the hardware organization of the system. MSC specifications do not explicitly specify features of the hardware on which processes communicate. Nevertheless, they represent hardware communication channels. We have noted in past work that a large number of assumptions concerning the properties of these channels are under-specified in MSC specifications [19, 21]. Hence expressing the physical structure component of the system architecture is not a strength of MSC specifications[4].

3. *Call structure*: Z.120 MSCs do not provide for representing call structures, although introduction of remote procedure call primitives is currently under study in the responsible ITU-T standardization committee. However, the Sequence Diagram notation as it is used in UML [9] allows for expressing procedure invocations in other, concurrent objects.

[4]We are currently working on extending the MSC notation to represent communication channels and their properties explicitly [10].

4. *Data flow*: MSC specifications represent data flows as message flows. The "*may send data to*" relationship of [3] corresponds to the *coordination relation* as we defined it in [5]: any pair (p_i, p_j) of processes (for $i \neq j$) in the MSC specification will be part of this relation iff there is at least one message sent from p_i to p_j in at least one of the bMSCs referenced in the HMSC.

5. *Control flow*: bMSCs define a strictly sequential control flow inside the processes. Composition via HMSCs renders the control flow per process potentially branching and iterating.

In conclusion, MSC specifications are strong in providing information about the system's process structure, data flow and control flow; they are weaker when it comes to representing conceptual and call structures; and they offer little support for expressing module, 'uses' and class structure.

3.2 Representing Architectural Content of MSC Specification in ROOM

When describing the architectural content of MSC specifications we will distinguish between *structural* and *behavioral*[5] components.

3.2.1 Structure

The structural component of the architectural content of an MSC specification will be represented in ROOM as follows:

- ROOM requires all component actors to be part of an enclosing actor. We introduce a *system actor* that does not have an explicit representation in an MSC specification. An MSC specification implicitly defines a "system". This system corresponds to the system actor.

- Every concurrent process in the MSC specification will be represented by exactly one concurrent ROOM *actor*[6]. As of now, MSC specifications do not express hierarchical nesting of processes, hence the resulting actor structure in ROOM is flat.

- For every pair of distinct processes (p_i, p_j) that is member of the coordination relation there will be the definition of exactly one ROOM *protocol*. In the definition of the protocol, one of the two processes will arbitrarily be chosen to represent the sending side of the communication, and all messages that it sends in all bMSCs to the other process will be part of the *out* signal list, the received messages form the *in* signal list.

- Each protocol that has been defined for each process p_i will result in an *end port* attached to the corresponding actor. If and only if the actor has been picked as the receiving side of the communication in the previous step the resulting port will be *conjugated* (essentially, *in* and *out* lists will be swapped).

[5]Note that [3] argue that "*the behavior of each component is part of the architecture, insofar as that behavior can be observed or discerned from the point of view of another component.*"

[6]Note that consistency requirements for MSC specifications mandate that every bMSC in an MSC specification consists of the same set of processes. MESA checks this as well as various other syntactic sanity constraints automatically.

- To represent the data flow in MSC specifications (which, in MSC specifications, is entirely message flow) we use ROOM *bindings* between ports. Note that ROOM only allows bindings between ports representing a base protocol and its conjugate.

3.2.2 Behavior

In ROOM, behavior can be expressed using hierarchical communicating extended finite state machines called ROOMcharts.

- MSC specifications describe sequences of send and receive communication events. This is expressed through the strictly sequential control flow of each of the concurrently composed processes in an MSC specification. Control passes from one local state of a process to the next one when executing state transitions. The state transitions can either be caused by *send* or by *receive* events. Hence, to represent this architectural feature we map local control states of the bMSC processes to states in ROOMcharts.

- We represent messages in bMSCs by ROOM *signals*. It is straightforward to map MSC receive events to trigger events inside ROOMcharts. Reception of signals is the only possible trigger for a ROOMchart transition. MSC send events are mapped to ROOM `SEND` primitives. These send out signals through a specified port and they may be executed in the course of a ROOMchart transition.

- There will be no ROOMchart associated to the system actor that we defined above, but every other actor will have a ROOMchart associated with it.

Note that MSC specifications represent a significant abstraction from the actual system behavior: 1) they represent send and receive events exclusively, no internal computation (we omit consideration of "internal events" and timer events from Z.120 MSCs), and 2) they represent message types only, no message parameters or data.

When representing the behavior in state machine transitions, inevitably the question arises as to how many events should be executed during one transition. Note that ROOM requires ROOMchart transitions to be triggered by `RECEIVE` events exclusively[7]. This entails that a transition starts with either a `RECEIVE` event as specified in the MSC specification, or with an auxiliary `RECEIVE` event that may be caused by self-sending of a message in the previous transition, or by the expiry of a timer.

As an illustration, consider process `CONTROL` in Figure 2: reception of message `START` in bMSC `START` is certainly considered the trigger for a corresponding ROOMchart transition. But where is this transition to terminate? We see the following possibilities:

1. The transition terminates after executing exactly one event, i.e. the ?`START` event (we use '?' to denote a receive, and '!' to denote a send event). In this case, the next transition would have to be triggered by an auxiliary `RECEIVE` event before executing !`START_ACK`.

[7]These `RECEIVE` events may originate from message exchanges or from expiring timers. Note that ROOM does not know "spontaneous" transitions as they can be found in SDL-92 [24] as well as many process algebras.

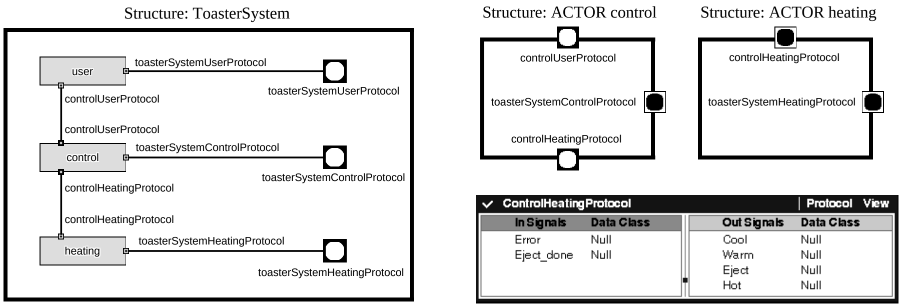


Figure 5: Structure of synthesized Toaster ROOM model

2. The transition terminates before the next receive event, or whenever the last event in a bMSC is encountered. In this case the above mentioned transition would comprise events !`START_ACK` and !`HOT`.

3. The transition executes all send events until the next receive event is seen. If we follow the path in the HMSC graph from bMSC `START` via `TOAST` to `EJECT`, then the transition could comprise the event sequence !`START_ACK`, !`HOT`, !`TOAST_ACK`, !`WARM` and !`COMREQ`. This interpretation clearly has the advantage that we do not need to introduce auxiliary trigger events. Conversely, for pathological MSC specifications where one process would never receive an event this could lead to divergent behavior. In previous work we have defined algorithms to syntactically detect such process divergence [5].

We will discuss advantages and disadvantages of the above choices in Section 4.

4 Synthesis Algorithms

Following the structure of the two major building blocks of the ROOM language our synthesis algorithm proceeds in two steps: first, the *structure* of the ROOM model will be generated. The *behavior* of the structural elements is generated in the second step. We discuss the synthesis algorithms and illustrate their use with an example. Pseudo-code representations of the algorithms are given in Appendix B.

4.1 Structure

The structural elements in ROOM comprise actors, protocols, ports and bindings. A pseudo-code level formalization of the algorithm that we use to synthesize structure is presented in Appendix B.1. The algorithm has the following features:

Actors: As we argued earlier, the algorithm creates exactly one actor for every process in the MSC specification, plus a system actor. Figure 5 shows part of the structure for the Toaster system as automatically generated by the Mesa toolset. At the center of the structure are the three actors `user`, `control` and `heating` that form the `ToasterSystem`. Note that the

ToasterSystem actor has been refined into the actors `user`, `control` and `heating`, as also shown in Figure 5.

Protocols: The algorithm analyzes the coordination relation of the MSC specification and generates a pair of protocols for every pair of processes that is part of the coordination relation. The protocol consists of a set of *in* and a set of *out* signals. To compute the coordination relationship as well as the signal lists the synthesis algorithms performs a depth-first search on the HMSC graph. When dereferencing an HMSC node the referenced bMSC will be analyzed and it will be determined which processes communicate with which others, if there are new communication relationships these will be added to the coordination relation. Further, it will be recorded which messages will be exchanged for which communication relationship. Note that a consistency requirement for MSC specifications mandates that every bMSC must have the same set of processes.

In the TOASTER example the algorithm detects the following coordination relation: {*(user, control), (control, heating)*}. Note that there is no communication relationship between *user* and *heating*. Figure 5 depicts an ObjecTime protocol editor window showing the `ControlHeating` protocol as derived from the `Toaster` MSC specification. It is used by the `control` actor to communicate with the `heating` actor.

Ports and Binding: For each pair of processes in the coordination relation there will be a pair of ports on the corresponding pair of actors. Ports are labeled with the names of the protocols that they are using. Pairs of ports corresponding to pairs of processes in the coordination relation will be connected by a ROOM binding. One of the processes in each of these pairs will be selected and designated as the conjugate port. In Figure 5 the line between actors `control` and `heating` represents such a binding. In the detailed structure of actors `control` and `heating` it can be seen that the ports labeled `controlHeatingProtocol` have different graphical appearances – this indicates the conjugation of the associated protocols. The `ControlHeating` protocol for the `controlHeatingProtocol` port in actor `heating` has the list of signals of in and out signals inverted compared to the `ControlHeating` protocol for the `controlHeatingProtocol` port in actor `control`, which is depicted in Figure 5.

System Actor: Finally, an enclosing system actor will be generated, c.f. the `ToastSystem` actor in Figure 5.

We have not yet discussed the meaning of the end ports inside the `ToasterSystem` actor in Figure 5. These are motivated by implementing branching choices in the HMSC graph. We will explain their meaning in the next Section when we discuss behavior synthesis.

4.2 Behavior

Earlier on we discussed three alternatives for the termination of transitions. We immediately dismiss the first option which suggested to terminate a ROOMchart transition after every communication event. The resulting ROOMcharts would have many transitions that rely on auxiliary triggers to model send events in the MSC specification. Hence, the first option would yield very complex ROOM models that are in the danger of blurring the designers intentions. This leaves us with

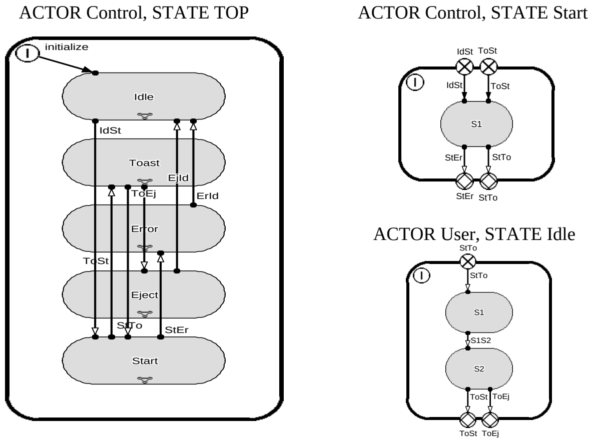


Figure 6: Parts of synthesized behavior of actors `Control` and `User`,based on maximum traceability algorithm

the second and the third options, and we suggest two different behavior synthesis algorithms to implement these alternatives:

Maximum Traceability: This algorithm implements the second termination alternative, i.e., a transition is terminated whenever a receive event inside a bMSC is encountered, and at the very latest at the end of the bMSC in which the transition is triggered.

Maximum Progress: This algorithm terminates a transition whenever the "next" receive event is encountered. The next receive event does not have to be encountered in the same bMSC in which the transition was originally triggered. Finding the "next" event involves a traversal of the HMSC graph in oder to find a reachable bMSC in which a receive event is found. I.e., we allow the state machine to progress as far as possible in the HMSC graph before we terminate the transition with the next receive event.

There are two major differences between these algorithms. The maximum progress algorithm yields potentially smaller and hence more efficiently executing models. However, let us assume that the HMSC graph of an MSC specification expresses certain high-level design choices concerning the decomposition of the system into phases of operation. Hence, the maximum traceability algorithm provides a state machine that captures the designers intention with respect to the phase decomposition much more faithfully than the maximum progress algorithm, but the resulting model may execute less efficiently. We will proceed with the discussion of both alternatives. For both algorithms we assume that the structure is already synthesized as described above.

4.2.1 Maximum Traceability Algorithm

The objective of this algorithm is to preserve traceability of components of the synthesized ROOM architecture with respect to the structure of the original MSC specification. For every bMSC referenced in the HMSC, and hence for every actor in the synthesized ROOM model, the following steps are performed:

```
define StEr
   from state S1
   to border transition StEr
   triggers
       {
       define signals {Error} on {controlHeatingProtocol};
       };
define StTo
   from state S1
   to border transition StTo
   triggers
       {
       define signals {Dostto} on {toasterSystemControlProtocol};
       };
```

Figure 7: Synthesized transition to resolve non-local choice in ROOM linear form

- All nodes of the HMSC graph are translated into top level states of the ROOMchart associated with each actor. The start node is converted to the initial point in the top level ROOMchart. Every interior node of the HMSC graph is mapped onto a state in the top level ROOMchart, and the name of the respective bMSC will be used as the name of the state it is associated with. Consider the top level state of the ROOMchart of the synthesized actor `control` in Figure 6 which consists of 5 states, each of them corresponding to one bMSC of Figure 2. Nodes in an HMSC graph represent instances of the bMSC type that they refer to by name. Hence, when an HMSC refers to more than one instance of any given bMSC type, the algorithm will use renaming to disambiguate the different occurrences of this bMSC type.

- All connecting edges of HMSC graph are converted into transitions in the top level ROOMchart. Since edges in the HMSC graph do not have names, the corresponding transition names are created using their source and destination state names, c.f. Figure 6.

Concludingly, the HMSC graph structure will be mapped onto a ROOMchart top level state machine so that bMSCs in the HMSC graph correspond to states and HMSC edges correspond to top level state transitions. This choice entails preservation of the high level design structure, but it also means that the state machines that refine the top level states must have transitions that terminate at the boundary of a bMSC. We will next consider how the states of the top level state machine will be implemented.

- The algorithm creates a basic ROOMchart for each bMSC reference in an HMSC. For example, consider the state `Start`, corresponding to bMSC `START` in the TOASTER example as depicted in Figure 6. For every node in the HMSC graph, each incoming HMSC edge will be mapped onto exactly one incoming transition point in the corresponding lower level ROOMchart. In Figure 6 the state `START` has two incoming transitions, one labeled `IdSt` representing the (`IDLE`, `START`) edge and another one labeled `ToSt` representing the (`TOAST`, `START`) edge edge of the HMSC graph, respectively.

  For each bMSC node, the algorithms performs the following:

1. For every process, it compiles a message list *ML* which comprises all communication events that this process executes in this bMSC. The entries are in the order of the events happening in the bMSC. The elements in the list are tuples of the form *(S, T, A, D)* where S is a local control state of the process, T is a triggering receive event, A is an action, i.e., a send event, and D is a local successor state of the process.
2. The algorithm proceeds with construction of the basic ROOMchart. The first element of ML will be analyzed. If it corresponds to an entry that has a non-empty T element, i.e., it corresponds to a receive event, then a transition to a state D will be compiled. This transition comprises all entries in the list until either another entry corresponding to a receive event or the end of the list will be encountered. Compare the `Start` state in Figure 6, which consists of just one state, i.e. it terminates at the end of the ML list, with the `Toast` state.
3. If the first message in a bMSC is a send message, then the generic timeout trigger T is used instead of a message receive event. The periodic timer T will be set to the smallest possible time step value in the initial transition of the Top state of every actor.
4. As a result of the above operations we obtain a temporary ordered list consisting of *(transition, newstate)* pairs. The first element in the list corresponds to the entry transitions into this basic ROOMchart, and we connect copies of this transition with the first *newstate* that is encountered in the temporary list. C.f. the `IdSt` and `ToSt` transitions in Figure 6 which connect to the respective incoming transition points. The *newstates* in the temporary list are converted to states in the basic ROOMchart. They are called $S_1, S_2, \ldots.$ The transition names are created by combining the first two letters of the source and destination states names.
5. The final *newstates* in the temporary list are connected to all outgoing transition points of this basic ROOMchart, c.f. transitions `StEr` and `StTo` in state `Start` in Figure 6.

- Transtions that involve a change of the high-level ROOMchart can be decomposed into components, as depicted in Figure 8. A branching in an HMSC translates into a change in the corresponding ROOMchart. The choice with which ROOMchart to continue will be made in the first of the three part transition. As a consequence, events that decide about a decision may be moved up to the predecessor state. As an example, the reception of message `Start` in state `Start` of actor `Control` in Figure 6 will be moved up to states `Idle` and `Toast`.

- ROOM does not permit the definition of triggers on initial transitions. Therefore, if a state in the high-level ROOMchart is the successor of the initial point and the first message of the corresponding bMSC is a receiving message, then an auxiliary state will be added, c.f. Figure 6, the state `S1` of actor `User` in state `Idle`.

- Decisions in MSC specifications can either be local or non-local.

  Local Decisions: We represent local decisions as non-deterministic choices of the actor that is sending the first message in the corresponding MSC choice. All other actors follow in the direction that the first actor has chosen. In our synthesized model these decisions could be either obtained by a random-number based algorithm, or by an interaction with the environment, i.e. the user running a simulation. We have chosen the latter solution in our synthesis. In Figure 5, which represents part of the structure of the ROOM model synthesized from the TOASTER specification, we introduce auxiliary end ports inside the system actor, e.g. `toasterSystemControlProtocol`. They permit the injection messages guiding the connected actors in their decision making. Suitable control message protocols will also be synthesized.

Non-local Decisions: Non-local choices require more algorithmic overhead for their solution. In [21] we suggested an algorithm based on synchronizing history variables to resolve non-local choices. However, we believe that this is not appropriate for our purposes. Therefore, similar to the solution of the local choice we leave the synchronization of the two processes up to a user in the environment of the system. Consider the process `Control` in bMSC `Start` of Figure 2: Assume that `Control` is the first process to reach the branching point between bMSCs `ERROR` and `TOAST` in the corresponding HMSC. Because all processes are concurrent, the outside world now has to make the decision as to where `Control` should proceed. In the synthesized behavior model this corresponds to the decision whether when `Control` is in state `Start`, sub-state `S1`, it will proceed via the transition sequence `StTo - StTo` into state `Toast`, or whether it will proceed via `StEr - StEr` into state `Error`. The code for the transitions `StTo` and `StEr` in Figure 7 shows how this is resolved: If first a message `Error` arrives, the `StEr` transition will be taken. However, if the user first injects a message `Dostto` through the `toasterSystemControlProtocol` port (c.f. Figure 5), then it will proceed with the `StTo` transition which ultimately leads into state `Toast`.

The apparatus devoted to dealing with non-local choices shouldn't be over-valued. The non-local decisions are part of syntactically legal MSC specifications, and therefore we have to deal with them. Non-local choices represent underspecified process synchronization in MSC specifications which has to be rectivied during the design process.

- In the synthesis of ROOM models, there are a few more issues that need to be considered:

Naming/labeling: ObjecTime has certain implementation constraints pertaining to the choice of names of states, state transitions and actors in a ROOM model. These force us to do renamings of various sorts in the synthesis.

Deadlocks: Even though we have shown Z.120 compliant MSC specifications to be deadlock-free [5] certain communication resource assumptions can lead to blocking implementations of MSC specifications. Consider a model where every process possesses exactly one input queue for all incoming messages, as this is assumed in ROOM as well as in SDL. Now, consider that this process receives messages `a` and `b` from two different processes and that the respective receive events are directly adjacent in the MSC specification, e.g. `?a` has `?b` as direct successor in this processes *ne* relation. Assume that the two processes do not synchronize their sending, then they may decide to first execute `!b` followed by `!a`. This means that the receiving process will see a message of type `b` while expecting a message of type `a`. Assuming a blocking receive semantics this results in a blocking specification. [2] call this situation a potential hazard and provide algorithms

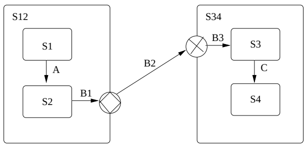


Figure 8: Transition in ROOM involving hierarchical states

for syntactic detection of this feature in a bMSC. These algorithms could easily be implemented in MESA. Also, the Poga tool developed at Bell Labs [13] implements these analysis algorithms.

Silent processes: We call a process, that does neither send nor receive a message in a bMSC, a *silent process* within this bMSC. Evidently, this process does not require a state synthesized for this bMSC. We offer two different approaches towards dealing with silent processes:

1. There is no transition and no state within the basic ROOMchart corresponding to the bMSC with the silent process. All incoming transitions that correspond to an entrance into this bMSC will be connected to successor states as indicated by the HMSC graph.
2. The state in the high-level ROOMchart is preserved and the state corresponding to the empty silent bMSC will be a dummy state consisting of one state and an empty transition. To simulate the empty transition, a timeout trigger event will be used.

In our implementation we chose the second alternative as it seems to better preserve the structure of the corresponding HMSC graph.

Message Overtaking: Message overtaking occurs in a bMSC when two message arrows between the same two processes in the same direction cross over[8], see Figure 9. As [11] argue, this has bearing on the communication model that an underlying system must have. In particular, according to [11] one message overtaking one other message requires at least two communication channels between the two processes. This means that in Figure 9 there would have to be distinct channels for transmitting message `data` and message `reset`. However, the ROOM semantics is defined such that any actor has only one input buffer for all incoming traffic, and it follows that we cannot synthesize executable ROOM code for MSC specifications that feature message overtaking.

After all necessary data structures have been built up, the ROOMchart behavior specifications in ROOM linear form will be stored in files for later import into ObjecTime Developer.

4.2.2 Maximum Progress Algorithm

The motivation for this algorithm is to determine maximum progress transitions in the MSC specification and to map these onto ROOM behavior descriptions. This means that synthesized ROOMchart transitions can span events originating from more than one bMSC. Hence, we do not use a

[8]Note that Z.120 mandates that message arrows may only be either horizontal or downwards facing.

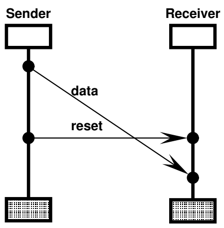


Figure 9: bMSC with message overtaking

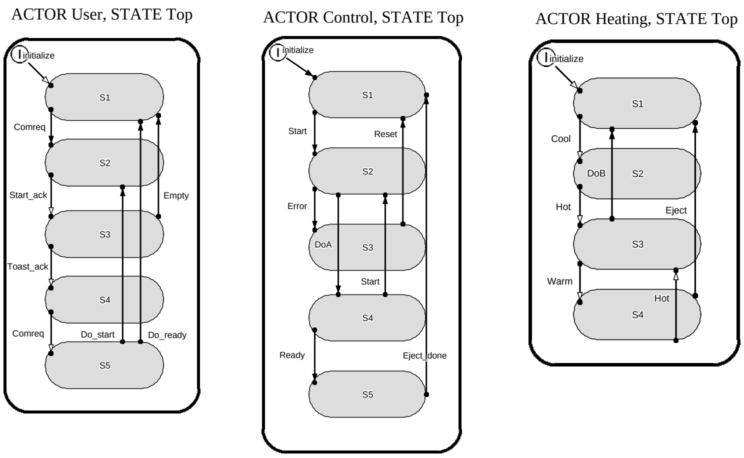


Figure 10: Behavior of Toaster model, synthesized by maximum progress algorithm

hierarchical state machine structure, the synthesized ROOMcharts will be flat. One ROOMchart per process in the MSC specification will be generated, see Figure 10. The maximum progress in every process will be different, hence the corresponding state machines will not have identical structure.

The algorithm proceeds in three steps. For each process P_i the following steps will be performed:

1. A graph of all messages contained in all bMSCs for which P_i is either a sender or a receiver is built. The graph is called the message graph reminiscent of the *ne* components of the message flow graphs in [20]. The nodes of the graph are similar to items of the message list (ML) of the previous algorithm. However, the message graph covers more than one bMSC and may be iterating and branching as determined by the HMSC graph. The message graph imposes a total order for all communication events and defines the causal relation amongst them. This construction is similar to the "unfolding" construction described for iterating and branching Message Flow Graphs in [20].

2. The message graph of a process is used to find a set of traces ($T = t_1, t_2, ..., t_k$). Each member of T, $t_k = (SourceState, Transition, Action, DestinationState)$. The members of T collectively describe all the states ($SourceState$ and $DestinationStates$) that a process can be in, transitions ($Transition$) which take it from a state to another, and all the actions (*Action*) that the transitions may cause. The transitions and actions correspond to the receive and send events respectively. A transition (receive message) may cause sending of zero, one or more messages.

   (a) The process P_i is initially assumed to be in state S_0. The subsequent states are numbered increasingly unless it is an old state (corresponding to a loop in the message graph). The first transition is the default transition which is defined for all ROOMcharts and is called *Initialize*. If the first message in the message graph is a receive message, the default transition (*Initialize*) will have no action associated with it. In this case the first trace will be $t_1 = (S_0, Initialize, Null, S_1)$ indicating that the first trace is from state S_0 to state S_1, which are connected by transition $Initialize$ which causes no action ($Null$). While the message processed is not a receive, the first transition (*Initialize*) is used to send messages as its action. For example, the *Initialize* transition in the top ROOMchart of *User* and *Heating* in Figure 10 will cause them to go state to S_1 and wait for the next message `Comreq` and `Cool`, respectively. Whereas, the *Initialize* transition of actor *Control* results in sending of `Comreq` and `Cool` messages.

   (b) For every receive message, a new trace is written with the corresponding receive message as its transition. The action items of the trace are found by searching the message tree depth-first to find all the send messages which are caused by this receive message. The search is continued till for every succeeding branch a receive message is encountered. All the send messages found in the meanwhile are assigned as the action items of the transition being processed.

3. The Trace set T is converted to ROOM specification. State S_0 is translated into the initial point in the ROOMchart. The subsequent state elements ($SourceState$ and $DestinationStates$) of the trace are used to create new states for the ROOMchart. Similarly, $Transition$ and

Action parts are translated into trigger event of the transitions and the code for sending of the corresponding signals, respectively.

4.3 Implementation in Mesa

The algorithms that we have described so far have all been implemented within the Mesa toolset. Mesa maintains its own internal data structures to represent MSC specifications which the synthesis algorithms directly access. Mesa writes the synthesis results (both structure and behavior) in ROOM *linear form* [22, 27]. An ObjecTime *package* will then be generated and it can be used to read all linear form files into ObjecTime Developer. ObjecTime allows transition code, which in our case only consists of code to send messages, to be provided in either C++ or Rapid Prototyping Language (RPL) code. In consideration of its usefulness at later stages in the design process we have chosen to generate C++ transition code.

Mesa consists of approximately 13,000 lines of C++ code, plus 2,000 lines of Tcl/Tk code for the graphical user interface. The synthesis algorithms add another 6000 lines of C++ code.

5 Use of Synthesized Models

In previous sections we presented parts of the synthesized ROOM model for the Toaster example. At this point we want to discuss the questions of what the synthesized models can be good for. [3] contains an extensive list of potential benefits of using architectural models, we just want to discuss a couple of these.

First, [3] emphasize that it is important to have software architectures that designers can "play around" with. Executability is therefore one of the benefits of a behavioral architecture description in a language like ROOM. Figure 11 illustrates how the executability of ROOM models can be exploited by the simulation system that ObjecTime Developer provides. Figure 11 shows the real-time simulation of the synthesized Toaster model. The structure browser (Figure 11, top right) allows the user to define points of observation and access points ("demons" and "injection points", respectively, in ObjecTime Developer terminology) that can be used to observe the the system as it either performs step-by-step or random simulation. State changes will be illustrated by highlighting of the current states and transitions between them. The most powerful tool to visualize the systems execution, though, is the possibility to visualize execution traces as bMSCs, c.f. Figure 11 at the bottom. The MSC traces are useful for visualizing execution sequences that are longer than the bMSC scenarios in the original MSC specification and therefore provide a better overview and understanding of the system. Also, necessary user interactions to resolve local and non-local choices can be visualized, e.g. the message `Dostto` in Figure 11. Concludingly, executable specifications can be helpful in supporting communication and education of new team members ([3]).

Second, [3] argue that architectural models can help in evolutionary prototyping. As an example, the MSCs that ObjecTime Developer generates can be stored in Z.120 textual representation which can be parsed by the Mesa toolset. Hence, the designer can modify the synthesized model, execute a number of scenarios, and feed each one of them back into the world of MSC specifications. Also, ObjecTime Developer is capable of generating target system C++ code or code skeletons, which means that we can automatically translate high-level design descriptions into code skeletons.

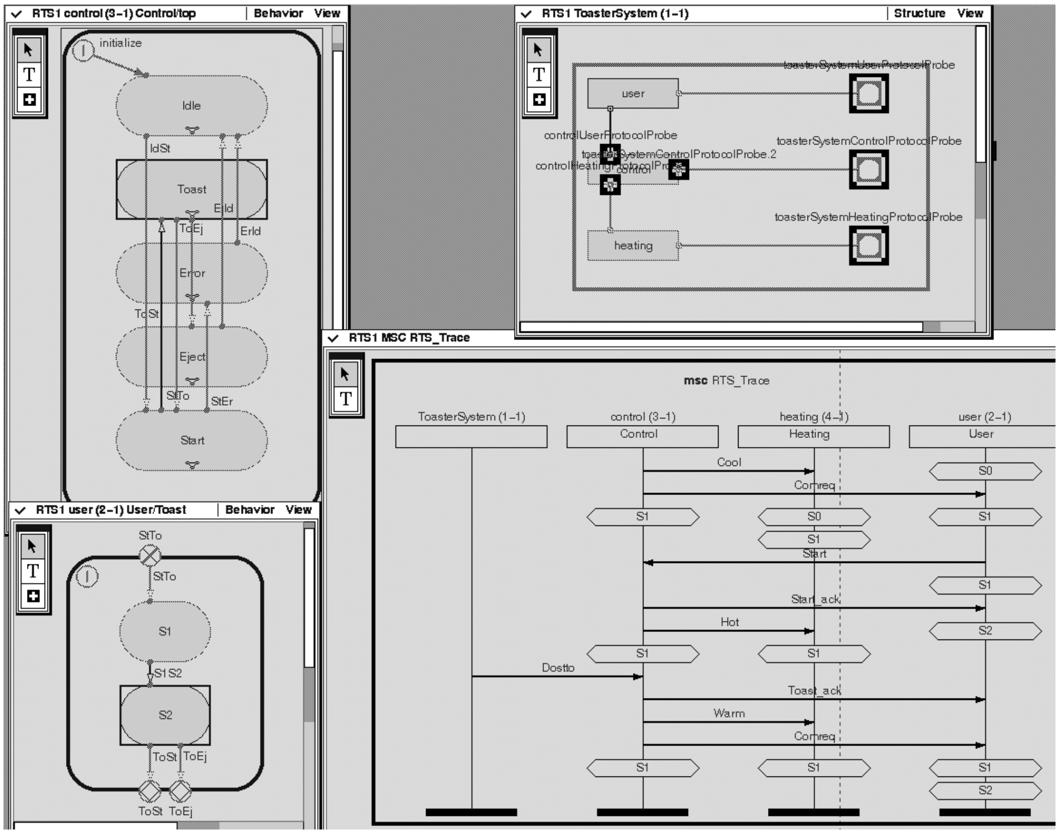


Figure 11: Simulation of synthesized ROOM model in ObjecTime

6 A complex example: GSM Mobility Management

To test the applicability of our algorithms and their implementation we have used the specification of a GSM (Global Systems for Mobile communications) Mobility Management protocol [23]. The mobility management in GSM keeps track of the mobile while on the move. Figure 12 shows a generic architecture layout of the GSM. The system consists of four main parts, MS (Mobile Station), BSS (Base Station Subsystem), MSC (Mobile services Switching Center) and Network. The MS is the portable unit used by the subscribers. The BBS controls the radio link with the MS. The MSC is part of the network subsystem but is separated from the rest of network to emphasis its role in the mobility management. The MSC manages the switching of calls between the mobile and other fixed or mobile network users. The Network in this example refers to the rest of the network which may consist of other land-based or mobile networks.

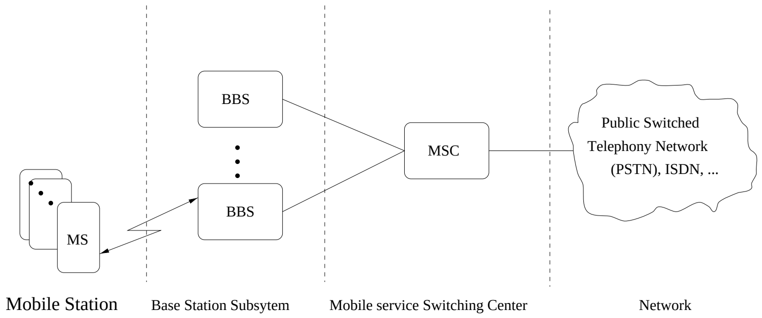


Figure 12: A generic representation of the GSM architecture layout

At any given time a mobile station (MS) is either switched off or on. While switched off the MS cannot be reached by the network. In on mode, an MS can respond to a paging (call) request, update its location or initiate a call. Figure 13 shows the set of services which are requested by the MS unit. Some other services such as call handover (transferring a call to a different cell) and roaming (to check if the MS is connected to the best possible channel) are not covered in this example. Each of the boxes in Figure 13 represent a bMSC which are given in Figures 14 and 15. Each bMSC shows the messages which are exchanged amongst the four processes.

Whenever an MS requires service, it sends a connection request to its BBS. In response the BBS assigns a channel to the requesting MS. These are modeled by `ConnReq` bMSC in Figure 14. At this point the MS may ask for the specific type of service it requires and they include initiating a call (`CallSetupReq`), responding to a call requested by another user (`PagingResp`) and or updating its location in the directory (`LocUpdReq`). These requests are then propagated to the the network through BBS and MSC which are handling the MS requests.

The network may ask the MS to identify itself. This may be the result of MS changing its MSC since the last time it accessed the system. Next, the authentication process is carried out. Its frequency is decided by the operator but in this implementation it has been made mandatory for all service requests. The authentication process is modeled by `Authenticate` bMSC. If the authentication is rejected, the MS needs to initiate another service request by sending a new connection request. Upon acceptance of the authentication (`Accept` bMSC), the network may ask for encryption (ciphering). This is an optional feature and may be added by the operator.

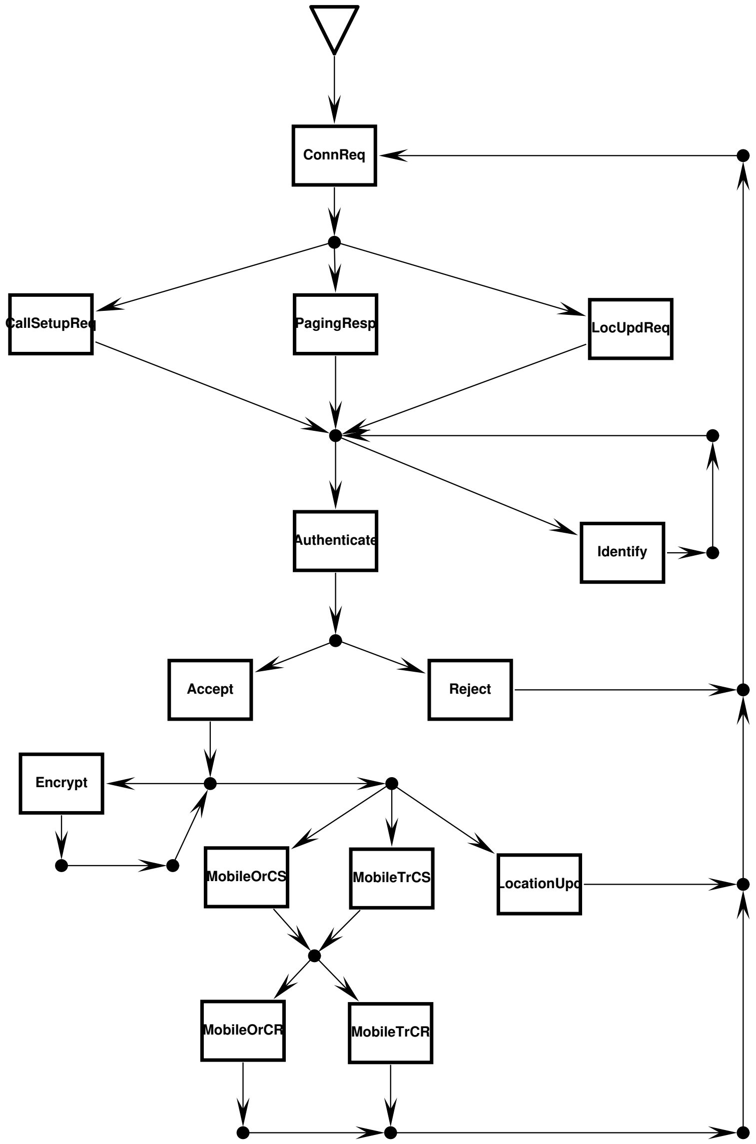


Figure 13: The HMSC specification of mobility management in a GSM network

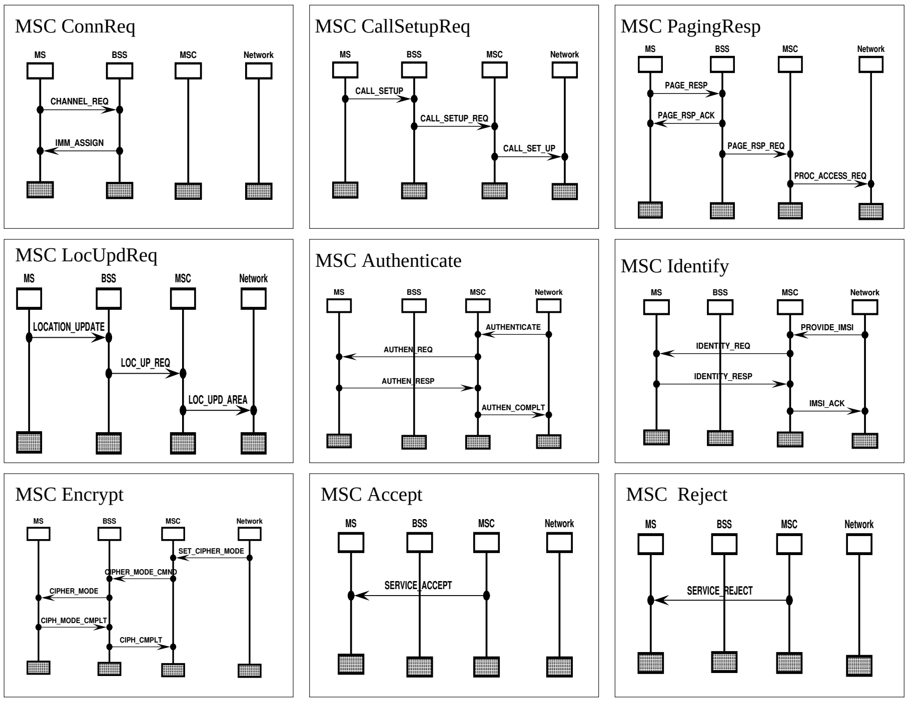


Figure 14: The bMSCs referenced in the HMSC in Figure 13

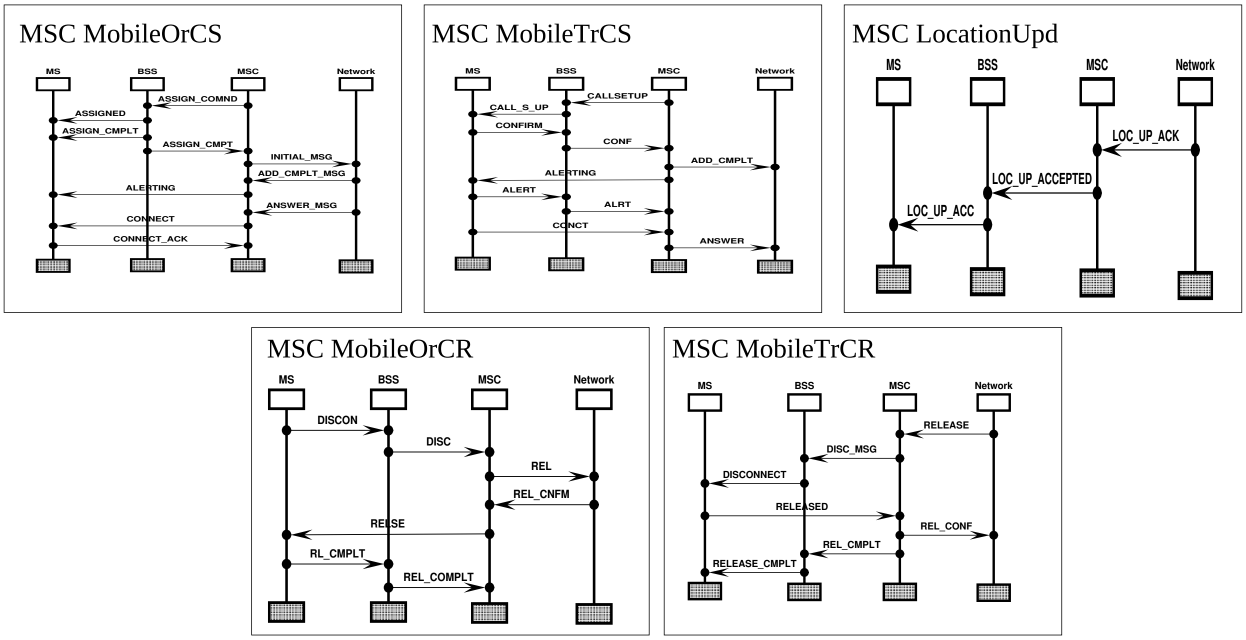


Figure 15: The bMSCs referenced in the HMSC in Figure 13 (Cont.)

The three bMSCs after `Accept` bMSC correspond to the acceptance of the network of the requests which were initiated by the MS at the beginning. For example if the MS had requested a location update `LocUpdReq`, the network after successful completion of identification (optional), encryption (optional) and authentication processes, would respond by initiating the `LocationUpd` bMSC. Similarly, `MobileOrCS` (mobile originated call setup) and `MobileTrCS` (mobile terminated call setup) bMSCs are initiated by the network in response to the `CallSetupReq` and `PagingResp` respectively.

Once a call is completed, it can be terminated at the request of the MS user or the other party. These are represented by the `MobileOrCR` (mobile originated call release) and `MobileTrCR` (mobile terminated call release), respectively.

The Synthesized Model

The structural synthesis algorithm produces five actors (four corresponding to MSC processes and one system actor) and seven protocols (four for the coordination relation in the MSC specification and three for injection points). The structure of the system actor `GsmSystem` is shown in Figure 16. It contains the other four actors and their bindings.

The maximum progress algorithm synthesizes the system behavior and creates four ROOMcharts. The ROOMchart for the MS actor has 11 states and 20 transitions. The ROOMcharts for BBS, MSC and Network actors have 9 states and 16 transitions, 17 states and 27 transitions, and 5 states and 15 transitions, respectively.

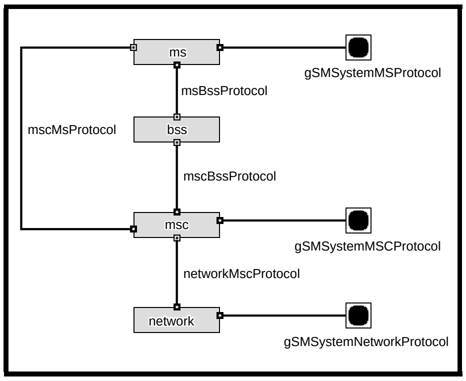
The maximum traceability algorithm produces identical top ROOMchart for all actors. These top ROOMcharts each have 14 states corresponding bMSCs in the HMSC and 32 transitions for the edges connecting the bMSC nodes. Figure 16 shows the top ROOMchart for the MS actor. Additionally, each actor has a number of states and transitions in its subROOMcharts. The subROOMcharts for MS, BSS, MSC and Network actors have 22 states and 71 transitions, 20 states and 69 transitions, 24 states and 79 transitions, and 18 states and 67 transitions, respectively.

The difference between the number of states and transitions produced by the two algorithms are quite significant. The additional number of states and transitions created by the maximum traceability algorithm is the trade off for preserving the hierarchical information contained in the MSC specification.

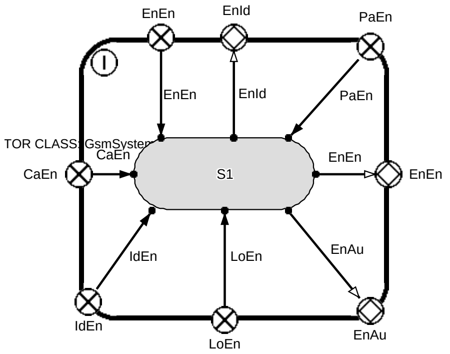
7 Conclusion

Summary. We presented algorithms which extract structural and behavioral architecture information from ITU-T Z.120 MSC specifications. The architectural information is represented in the ADL ROOM. The structural part mainly represents the concurrent processes and their communication relationship. For the behavioral part we have suggested two variants for the analysis: the maximum traceability algorithm, which emphasizes preservation of the HMSC structure in the state space of the synthesized model, and the maximum progress algorithm, which provides more succinct state machines but sacrifices traceability with respect to the HMSC graph. The algorithms have been implemented in the Mesa toolset that we are currently developing. Finally, we discussed aspects of the practical use of the synthesized models in the software design process. Overall, our work contributes to making Z.120 MSC specifications more useful in the Software design process.

ACTOR GSMSystem, Structure



ACTOR MS, STATE Encrypt



ACTOR MS, STATE TOP

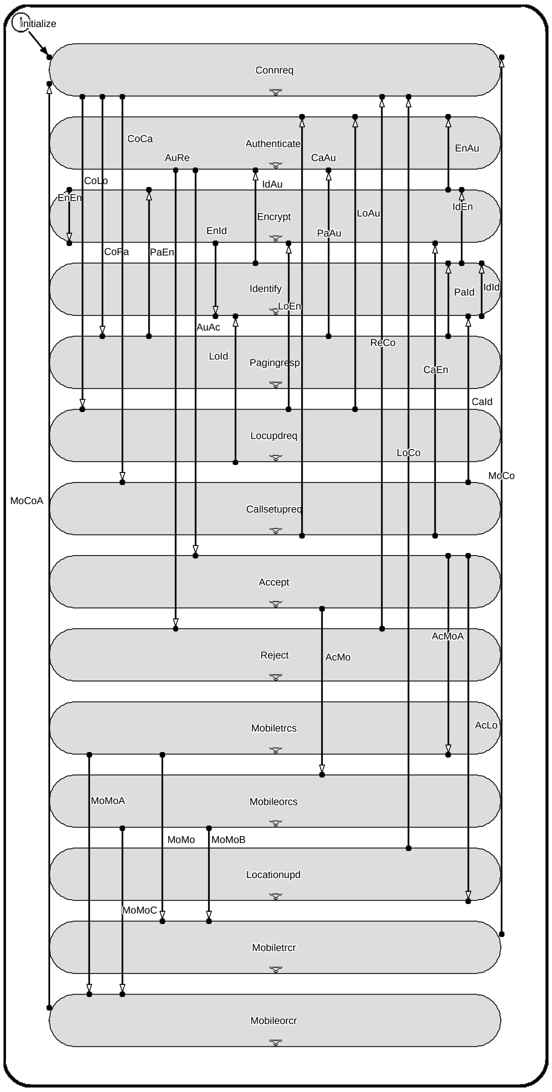


Figure 16: The synthesized models of the structure and behavior of the GSM example

Future research. The synthesis algorithms have been formalized, and their formalization can be found in Appendix A. However, we currently rely on an intuitive understanding of their functioning and have no proof of properties that they preserve. A formal investigation of this aspect is an open issue. Furthermore, we need to further investigate the use of the generated ROOM models in the design process, including guidelines for further refinement transformations. It should finally be noted that while we have found ROOM to be a very suitable design notation, our work is in principle also applicable to other, similar notations, like UML, UML-RT and SDL.

Acknowledgements. We wish to thank Bran Selic for constructive criticism on an earlier version of this work. Financial support for this research was provided by: ObjecTime Limited, Communications and Information Technology Ontario (CITO), and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] B. Algayres, Y. Lejeune, F. Hugonment, and F. Hantz. The AVALON project: a validation environment for SDL/MSC descriptions. In O. Faergemand and A. Sarma, editors, *Proceedings of the 6th SDL Forum, SDL'93: Using Objects*, October 1993.

[2] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055*, pages 35–48. Springer Verlag, 1996.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley Longman, 1998.

[4] H. Ben-Abdallah and S. Leue. Mesa: Support for scenario-based design of concurrent systems. Technical Report 97-12, Department of Electrical & Computer Engineering, University of Waterloo, October 1997 1997. 17 p.

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

[6] H. Ben-Abdallah and S. Leue. Timing constraints in message sequence chart specifications. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X / PSTV XVII '97*, pages 91 – 106. Chapman & Hall, November 1997.

[7] H. Ben-Abdallah and S. Leue. Mesa: Support for scenario-based design of concurrent systems. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 4th International Conference*, volume 1384 of *Lecture Notes in Computer Science*, pages 118 – 135. Springer Verlag, 1998.

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

[9] Rational Software Corporation. UML notation guide. Research report, 1997. See also `http://www.rational.com/uml`. Also The Object Management Group, document number ad/07-08-05.

[10] R. Delon. Implementation in mesa of the PROMELA code generator. E&CE 499 Project Report, Univesity of Waterloo, April 1998.

[11] A. Engels, S. Mauw, and M. 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, FORTE X/PSTV XVII '97*, pages 75 – 90. Chapman & Hall, November 1997.

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

[13] G. J. Holzmann. Early fault detection tools. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055*, pages 1–13. Springer Verlag, 1996.

[14] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki. SDE: Incremental specification and development of communications software. *IEEE Transactions on Computers*, 40(4):553–561, Apr. 1991.

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

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

[17] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi. On the role of scenarios in object-oriented software design. In Lars Bendix, Kurt Normark, and Kasper Osterbye, editors, *Proceedings of the Nordic Workshop on Programming Environment Research, Aalborg, May 29-31, 1996.* Aalborg University. published as WWW-URL `http://www.cs.auc.dk/ normark/NWPER96/proceedings/proceedings.html`.

[18] Kai Koskimies, Tarja Systä, Jyrki Tuomi, and Tatu Männistö. Automated support for modeling OO software. *IEEE Software*, 15(1):87–94, January-February 1998.

[19] P. B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques, VII*, Proceedings of the Seventh International Conference on Formal Description Techniques. Chapman & Hall, 1995.

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

[21] S. Leue and P. B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proceedings of the 2nd Workshop on the SPIN Verification System, Rutgers University, August 5, 1996.* American Mathematical Society, DIMACS/32, 1997.

[22] ObjecTime Limited. ObjecTime Toolset Guide, Document Version 1.0 for ObjecTime Developer 5.1, August 1997.

[23] Asha Mehrotra. *GSM System Engineering*, chapter 4. Artech House Publishers, Norwood, MA, 1996.

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

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

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

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

[28] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `www.objectime.com/new/uml/index.html`, March 1998.

[29] S. Somé, R. Dssouli, and J. Vaucher. From scenarios to timed automata: Building specifications from user requirements. In *Proceedings of the 2nd Asia Pacific Software Engineering Conference.* IEEE, December 1995.

A Formal Representation of the Synthesis Process

In this appendix we present the formal definitions of both MSC and ROOM specifications which were used in our synthesis algorithms. These definitions are used to map the various structural and behavioral components in the two specifications.

A.1 Formal Definition of MSCs (from [5, 20])

An *MSC specification* is a structure $M = (B, V, suc, ref)$ where:

- B is a finite set of bMSCs;
- $V = \top \cup I \cup \bot$ a finite set of nodes partitioned into the three sets of singleton-set of *start* node, *intermediate* nodes, and *end* nodes, respectively;
- $suc \subseteq (\top \cup I) \times V$ the relation which reflects the connectivity of the HMSC of M such that all nodes in V are reachable from the start node; and
- $ref : I \longmapsto B$ a function that maps each intermediate node to a bMSC.

Each intermediate node of MSC is called a reference node and is mapped into a bMSC. The bMSC is defined in [20] as a tuple $\mathcal{G} = (S, C, X, ne, sig, ST, stype, ET, etype)$ where S, C, and X denote arbitrary pairwise disjoint sets, the elements of which are called *sending* events, *receiving* events, and *extra* nodes respectively. Furthermore, ST and ET denote arbitrary disjoint sets (also disjoint from S, C and X), whose elements are called *signal* and *event* types. $(S \cup C \cup X, ne, etype, ET)$ is a digraph with node labels and $(S \cup C \cup X, sig, stype, ST)$ is a digraph with edge labels.

A.2 Formal Definition of ROOM Model

We omit formalizing those parts of ROOM we do not use in our synthesis. A ROOM model is a tuple $R = (A, P, Bind)$ where:

- A is a set of actors defined as a tuple $(A, Ports, RC)$ where:
  - A is the set of other actors which are contained in A.
  - $Ports : endports \rightarrow conj$ where $endports$ is the set of end ports and $conj = \{0, 1\}$ is the type of the port which is ordinary or conjugated.
  - RC is a ROOMchart which is defined below.
- P is a set of protocols defined as $P = (InSig, OutSig)$ where $InSig$ and $OutSig$ are the sets of input and output signals, respectively.
- $Bind \subseteq A \times A$ is the binding relation.

The behavior of each actor is given by a hierarchical extended finite state machine called a ROOMchart. A ROOMchart is formally defined as $RC = (St, SubRC, ref, Tr, Ac, t, a, I)$ where:

- $St = St_c \cup St_l$ is a finite set of states partitioned into the two sets of singleton-set of composite and leaf states, respectively.
- $SubRC$ is the set of sub-ROOMcharts defined as $(St, Tr, Ac, I_{tp}, I_{Ac}, O_{tp}, O_{Tr}, t, a)$ where St is a set of states, Tr is a set of triggering events (receive messages), Ac is a set of actions (including send events), I_{tp} is a set of incoming transition points, I_{Ac} is a set of input actions, O_{tp} is a set of outgoing transition points, O_{Tr} is a set of output triggering events (receive messages), t is the transition function defined as $t : (St \times Tr \rightarrow St) \cup (St \times O_{Tr} \rightarrow O_{tp})$, and a is the action function $a : (St \times Tr \rightarrow A) \cup (I_{tp} \rightarrow I_{Ac})$.

- $ref : St_c \longmapsto SubRC$ a function that maps each composite states into a sub-ROOMchart ($SubRC$).
- Tr is a set of triggering events (receive messages).
- Ac is a set of actions (including send events).
- t is the transition function defined as $t : St \times Tr \rightarrow St$.
- a is the action function $a : St \times Tr \rightarrow Ac$.
- $I \in St_l$ is the initial point (state).

A_s is the container actor called the system actor which contains all other actors and has no ROOMchart, $A_s = (A, Ports, Null)$.

B Algorithms

B.1 Algorithm for Structure Synthesis

Input: MSC specification $M = (B, V, suc, ref)$.
Outputs: The structural components of ROOM model $R = (A, P, Bind)$.

Create the system actor A_s
For all bMSC nodes Do
For one of the bMSCs
For each process, add an actor with the same name to the actor set A
place an instance of each actor in the system actor A_s
create a list of messages (ML)
For each message M_i in the message list ML
create a new protocol $Prot$ corresponding the source and destination processes of M_i
If ($Prot$ or $conjugate(Prot)$) $\notin P$
add $Prot$ to P
add an end port to the actor corresponding to source of $Prot$
add a conjugated end port to the actor corresponding to destination of $Prot$
create a binding corresponding to the source and destination actors in system actor A_s
If ($Prot \in P$ and M_i is a send) or ($conj(Prot) \in P$ and M_i is receive)
add M_i to $OutSig$ list of P
Else
add M_i to $InSig$ list of P
End

B.2 Maximum Progress Algorithm for Behavior Synthesis

Input: MSC specification $M = (B, V, suc, ref)$ and the structural components of ROOM Model $R = (A, P, Bind)$ obtained in structure synthesis.
Outputs: A single (non-hieracrhical) ROOMchart per actor.

/* First we form the list of traces $T = T_1, T_2, ..., T_s$ without the state information i.e., $T_k = (Transition, Action)$ for each process P_i. */
For each process p_i
create message graph (MG_i)
$Q = MG_i$ /* the set of the messages to be processed */
While there is a message in Q

```
    For each message $m_l \in Q$
        If ($m_l$ is a send event ($m_l \in S$)
            If ($m_l \notin T$ (set of traces))
                T = T + (Init, $m_l$);
            Else
                Q = Q - first message in Q;
        Else If $m_l$ is the last successor of first message in Q
            Q = Q - first message in Q;
        Else
            Q = Q - all successors of $m_l$;
            l = index of next successor of first message in Q;

If first message is receive
    T = T + (Init, Null);

For l = 1 to N (/*N = number of all messages in $Q$*/)
    If ($m_l$ is a send event, $m_l \in S$)
        Q = Q - $m_l$;
        continue;
    If (l = N)
        T = T + ($m_l$, Null);
        continue;
    U = all successors of $m_l$;
    while U is not empty
        For each $m_k \in U$
            If ($m_k$ is a receive event $m_k \in C$)
                If ($m_k$ is an immediate successor of $m_l$)
                    T = T + ($m_l$, Null);
                If ($m_k$ is the last successor of $m_l$)
                    continue;
                Else
                    U = U - $m_k$ and all its successors;
            Else
                T = T + ($m_l$, $m_k$);
        Q = Q - $m_l$;
/* Then we augment the trace list $T$ with state information so that we have
$T_k = (SourceState, Transition, Action, DestinationState)$ for each process $P_i$.*/
For each process $P_i$
    $N_t$ = Number of traces in T;
    I = 1; /* index of traces */

    while there is a trace $T_k$ with transition Init
        $T_k$ = ($S_0$, Init, action, $S_1$);

    For each trace $T_k \in T$ Do
        If transition of preceding trace of $T_k$ is Init
            $T_k.SourceState = S_1$
        Else
            $T_k.SourceState = PrevTransState(T_k)$ /* state associated with the preceding receive */
```

48. If there is a state associated with $T_k.Action$, $ActionState(T_k)$
49. $T_k.DestinationState = ActionState(T_k)$; /* is a loop */
50. Else
51. $T_k.DestinationState = S_I + 1$;
52. $ActionState(T_k) = S_I + 1$; /*associate the state with action*/
53. $NextTransState(T_k) = S_I + 1$; /* associate the state with next receive event (transition). */
54. I=I+1;
/* Now we can create the ROOMchart */
55.For each process P_i
56. set the initial point to be S_0
57. create states, transition and actions corresponding to those in trace list T
End

B.3 Maximum Traceability Algorithm for Behavior Synthesis

Input: MSC specification $M = (B, V, suc, ref)$ and the structural components of ROOM model $R = (A, P, Bind)$ obtained in structure synthesis.
Outputs: A two-level (hierarchical) ROOMchart per actor.
1. For each actor A_i in A
2. For each bMSC node I_j in HMSC
3. create a new state in St_c
4. For each bMSC node I_j in HMSC
5. For all successor nodes of I_j
6. create a new transition Tr connecting I_j and its successor
7. For each actor A_i in A
8. For each state St_j in St_c
9. get the bMSC corresponding to St_j
10. create a list of messages (ML)
11. For each message M_k in (ML)
12. If M_k is a receive event
13. create a new trace T and assign M_k as its $Transition$
14. Else
15. create a new trace T and assign M_k as its $Action$
15a. assign a timeout signal as $Transition$ of trace T
16. For each T_m in T
17. If T_m has $Transition$
18. add a new transtion to Tr set of $SubRC_j$ (of bMSC I_j) corresponding to $Transition$
19. add a new state to St_l set of $SubRC_j$
20. Else
21. add a new action to Ac set of $SubRC_j$ corresponding to $Action$
22. For each incoming transtion point I_tp in $SubRC_j$
23. create a new transition Tr_k corresponding to first transition of $SubRC_j$
24. set the source of Tr_k to be I_tp
25. add Tr_k to Tr (transition set of $SubRC_j$)
26. remove the transition corresponding to first transition of I_tp from Tr
27. For each outgoing transition point O_tpl in O_tp in $SubRC_j$
28. create a new transtion Tr_k
29. set the destination of Tr_k to be O_tpl
30. add Tr_k to Tr (transiton set of $SubRC_j$

31.For each actor A_i in A
32. For each transition Tr_k in Tr of ROOMchart RC of actor A_i
33. move the first transition of the bMSC destination of Tr_k to
. the last transition of the bMSC source of transition Tr_k
34. If source of transition Tr_k is the Initial point and first transition of the bMSC
. destination of Tr_k is not a timeout
35. add a new state St_lk in state set St_l of $SubRC$ of the destination of Tr_k ($SubRC_n$)
36. create a new transition Tr_m in $SubRC_n$ and make it same as first transition in $SubRC_n$
37. set the source of Tr_m to be St_lk
38. create a new transition Tr_{m2} in $SubRC_n$
39. set the source of Tr_{m2} to be the incoming transition point corresponding to initial point
40. set the destination of Tr_{m2} to be St_lk
41. add Tr_{m2} to the set Tr of $SubRC_n$

/* To assign the injection points */
42.For each actor A_i in A
43. For each subROOMchart $SubRC_k$ of ROOMchart RC
44. If the number of successors of subROOMchart $SubRC_k$ is more than one
45. For each successor subROOMchart $SubRC_m$
46. Iff the first transition of $SubRC_m$ is a timeout signal
47. For each outgoing transition point of $SubRC_k$, O_tpm corresponding to $SubRC_k$
48. If O_tpm has a corresponding transition
49. create an injection signal
50. create an injection port and protocol if not existed before
End

C Linear Form

The linear form is the storage and data exchange format used in *ObjecTime Developer*. The only reference documentation that we could use were in the linear form directory of the toolset distribution which included a lex file and a simple parser that checks the syntax.

The best way to find out about the linear form was to create simple ROOM models in ObjecTime and to look at the generated output. By looking at many different files, most of the syntax could be figured out but the ROOMchart definition took a long time as it is done recursively.

The parser does not check name conflicts. Names define the semantics of the ROOM model. Thus the parser did not complain but ObjecTime Developer did not read the input created by MESA. Unfortunately, Developer does not provide any information about the possible problems and only syntax errors are displayed. We assumed name conflicts to be the reason i.e. states and transitions having the same name. After changing the algorithm to avoid same names, Developer read the linear form.

All necessary graphical information for the structure and behavior editor in Developer is written in the linear form. We had no access to the documentation about the graphical specifications. We managed to display the states and actors where we intended them to be but neither bindings nor transitions are displayed properly. Instead, Developer uses default values and puts the objects at the right end of the states or actors. This is due to a complicated graphic engine that calculates the coordinates of the boxes for the labels. As labels have variable lengths, our algorithm cannot calculate the right values. Thus the graphical information that is written in the files is of no use for ObjecTime Developer.

MESAmay be the first program to create linear form files other than the Developer itself. Two problems which we have observed but have not found reasonable explanations for them are:

1. ObjecTime Developer only reads the first model. It always complains when reading a second model no matter which. An exception occurs and an e-mail is automatically sent to ObjecTime Limited.
2. If the models reach a certain complexity, the compilation gets stuck. ObjecTime Developer indicates that it is still compiling but there is no way to interrupt it. Using the UNIX command `top`, one can see that ObjecTime Developer has forked another process that uses a lot of the CPU power and an increasing amount of main memory.

C.1 Object Classes

To write the linear form output, all necessary data has to be collected first. Then several changes have to be done before the information can actually be written to the files.

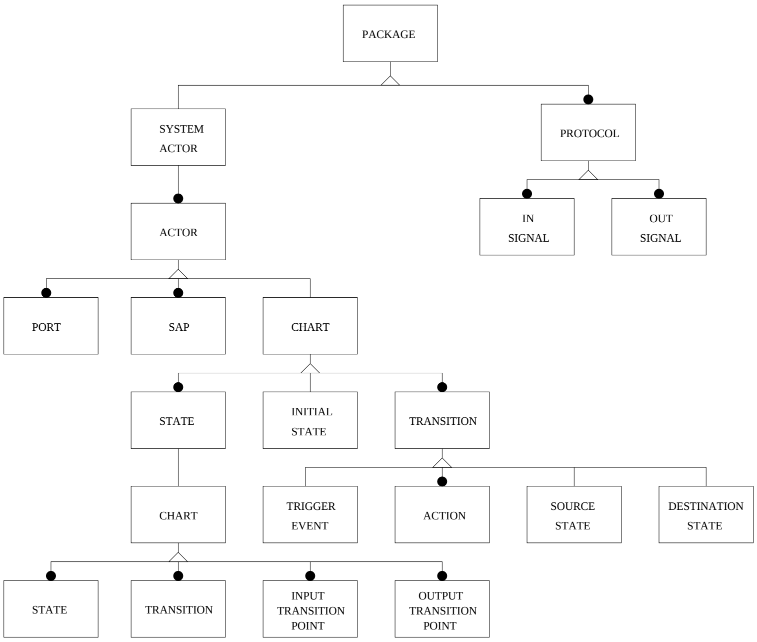


Figure 17: OMT diagram of object classes of the linear form

To do so, a subset of the internal class structure used in Developer is implemented inside Mesa. The result is shown in the OMT diagram of Figure 17.

The following paragraphs describe the classes used. The methods and the graphical information is omitted to better focus on the structure of the class hierarchy.

The root of the tree is the package class. Besides the package name and the path in the file system, this class includes references to the system actor class, the set of actors and the set of protocols.

```
class Package {
  public:
    char      name[packNameLength];
    char      path[pathNameLength];
    Actor*    system;
    Actor*    actors;
    Protocol* protocols;
};
```

Next there is the actor class. The actor is connected to information about the bindings and ports attached to it, as well as actors inside the actor and its ROOMchart. The actor class is used for two kinds of actors: the system actor and the actors derived from the MSC processes. The system actor uses the `endPorts` and `actors` references, whereas the actors use the `ports` and the `chart` references. The `saps` reference is only needed for the timeout service that is used to generate a trigger event during simulation. A common class is chosen because there is no distinction between different kinds of actors in ROOM either.

```
class Actor {
  public:
    char      name[actorNameLength];
    char      language[langlength];
    Protocol* bindings;
    Port*     ports;
    Port*     endPorts;
    Actor*    actors;
    SAP*      saps;
    Chart*    chart;
};
```

The `actor class` definition is followed by its components. First there is the protocol. A protocol is not directly linked to the actors. Instead, a port connects the actor to the protocols. They are defined by their source and destination actor and two sets of signals, one for the outgoing and one for the incoming signals.

```
class Protocol {
  public:
    char    name[protNameLength];
    Actor*  source;
    Actor*  dest;
    Signal* in_signals;
    Signal* out_signals;
};
```

A signal object consists of the signal name, source and destination actor plus the protocol it belongs to.

```
class Signal {
  public:
    char      name[sigNameLength];
    Actor*    source;
    Actor*    dest;
    Protocol* protocol;
};
```

The `port class` keeps track of the protocol and the direction of the signals. If two actors are linked by a protocol, then the ports have to be compatible. In other words, one of the two ports has to be conjugated so that all signals defined as incoming in one, are outgoing in the other one, and vice versa.

```
class Port {
  public:
    Protocol* protocol;
    int       conjugated; // != 0 -> conjugated
};
```

Actors can talk to other actors and they can talk to the real-time operating system as well. This is done using SAPs (service access points). SAPs are defined by their name, their type and the service requested. In this case, only the timeout service is used. The correct values are given in the code as a comment.

```
class SAP {
  public:
    char name[sapNameLength];     // e.g. MyTimer
    char type[sapNameLength];     // e.g. Timing
    char trigger[sapNameLength];  // e.g. timeout
};
```

At this point, the structural part is complete. The classes used from now on are dedicated to the behavior of the system. All actors (not the system actor though) have a behavior represented by a ROOMchart. A ROOMchart consists of a reference to its actor, a set of states and transition as well as a set of incoming and outgoing transition points. If the ROOMchart is inside of another ROOMchart, then a reference to the parent state is included, too.

```
class Chart {
  public:
    Actor*      actor;
    State*      parentState;
    State*      states;
    Transition* transitions;
    State*      inPoints;
    State*      outPoints;
};
```

States and transitions go hand in hand. States only know their name or number and a reference to a ROOMchart inside them, if there is one.

```
class State {
  public:
    char   name[stateNameLength];
    int    number;
    Chart* chart;

    hmsc_nodeT*  hmscNode;  // pointer to the hmsc node
    bmsc_graphT* bmsc;      // pointer to the corresponding bmsc
};
```

Transitions keep track of their source and destination states. In addition, they know their trigger event (either a signal or a SAP) and all the actions that are executed when the transition is fired.

```
class Transition {
  public:
    char    name[transNameLength];
    State*  source;  // source state
    State*  dest;    // dest state
    Signal* trigger; // trigger event (signal)
    SAP*    sap;     // trigger event (sap)
    Signal* actions; // list of actions
};
```

During the synthesis, another type of class is needed: the `trace class`. The trace class is used to create a list of signals extracted of the complex MSC data structure. The trace is then modified and converted into the ROOMchart information. A single trace list item contains the following information: Source and destination states (numbers only), the trigger and the actions.

```
class Trace {
  public:
    int     source;   // source state
    int     dest;     // destination state
    Signal* trigger;  // trigger event OR
    SAP*    sap;      // SAP as trigger event
    Signal* actions;  // list of actionsc
};
```

Looking at the data structure, one can see that the internal representation of ROOM models is rather complex and redundant. It is therefore unavoidable to collect all data first, modify it and write it to the linear form files as a last step. There is no way to create linear form output directly from the information given in the MSC structures.

C.2 The ASCII files

In this section we explain the ASCII files which are created as the result of the synthesis process. The `TOASTER` model was introduced in the beginning of this report, is used here for illustration. The algorithms produce the following files:

- The package file (`Toaster.package`)
- The system actor file (`ToasterSystem.actor`)
- The actor files (`User.actor, Control.actor, Heating.actor`)
- The system port files for the protocols between actors (e.g. `ControlHeatingProtocol.port`) and the protocols between the systemactor and the actors (e.g. `ToasterSystemHeatingProtocol.port`)

They are described in greater detail in the following paragraphs.

C.2.1 `Toaster.package`

The package file only lists the other files needed for the model. It is pretty self-explaining.

```
VERSION '1.1'

PACKAGE Toaster
  PUBLIC {
    ACTOR ToasterSystem;
    ACTOR User;
    ACTOR Control;
    ACTOR Heating;
    PROTOCOL ControlUserProtocol;
    PROTOCOL ControlHeatingProtocol;
    PROTOCOL ToasterSystemUserProtocol;
```

```
  PROTOCOL ToasterSystemControlProtocol;
  PROTOCOL ToasterSystemHeatingProtocol;
  };
```

Generally, ObjecTime Developer does not care about white space. The linear form is, however, case sensitive.

C.2.2 ToasterSystem.actor

The system actor class is the one that contains all other actors plus their bindings. But first the protocols between the system actor class and the other actors are defined.

```
VERSION '1.1'

ACTOR CLASS ToasterSystem
  INTERFACE {
    }
  IMPLEMENTATION {
    STRUCTURE {
      GOBORDER 56@40 AND 216@272
      ENDPORTS {
      DEFINE toasterSystemUserProtocol
        ISA ToasterSystemUserProtocol
        GOPORT 123@27 AND 149@53
          LABEL 84@53 AND 189@71 DEFAULT bottomCenter OUTSIDE;
      DEFINE toasterSystemControlProtocol
        ...
      DEFINE toasterSystemHeatingProtocol
        ...
        }
```

Next all the actors plus their ports are defined.

```
      COMPONENTS {
        DEFINE user
          ISA SUBSTITUTABLE User
          GOACTOR 96@72 AND 176@96
            LABEL 68@99 AND 173@117
            PORTS {
              controlUserProtocol GOPORT 117@93 AND 123@99
                LABEL 84@53 AND 189@71 DEFAULT bottomCenter OUTSIDE
                ANGLE 0.0;
              toasterSystemUserProtocol GOPORT 117@93 AND 123@99
                LABEL 84@53 AND 189@71 DEFAULT bottomCenter OUTSIDE
                ANGLE 0.0;
              };
        DEFINE control
          ...
        DEFINE heating
          ...
        }
```

The definition of the bindings is the next step. The protocol name is followed by the name of the actor (if not the system actor).

```
   BINDINGS {
     DEFINE B1 [*] BETWEEN controlUserProtocol/control
         AND controlUserProtocol/user
       GOBINDING 118@97 AND 123@120
         LABEL 111@99 AND 130@117
         FROM 120@99 TO 120@117;
     DEFINE B2 [*] BETWEEN controlHeatingProtocol/control
         AND controlHeatingProtocol/heating
         ...
     DEFINE B3 [*] BETWEEN toasterSystemUserProtocol
         AND toasterSystemUserProtocol/user
         ...
     DEFINE B4 [*] BETWEEN toasterSystemControlProtocol
         AND toasterSystemControlProtocol/control
         ...
     DEFINE B5 [*] BETWEEN toasterSystemHeatingProtocol
         ...
     }
   }
```

As a last step the behavior is defined. The system actor class has no behavior and therefore this section is almost empty.

```
  BEHAVIOR {
    LANGUAGE 'C++'
    FSM
      GOBORDER 56@40 AND 664@504
    }
  };
```

C.2.3 `Control.actor`

The control actor file is chosen as a representative of all actors. The linear form file starts with the definition of the interfaces.

```
VERSION '1.1'

ACTOR CLASS Control
  INTERFACE {
    PORTS {
      DEFINE controlUserProtocol
        ISA ControlUserProtocol
        GOPORT 123@27 AND 149@53
          LABEL 84@53 AND 189@71 DEFAULT bottomCenter OUTSIDE
          ANGLE 0.0;
      DEFINE controlHeatingProtocol
          ...
```

```
      DEFINE toasterSystemControlProtocol
          ...
      }
    }
```

Next there is the defintion of the structure. As the interfaces have already been defined above, there is only a reference to them in the definition of the endports.

```
  IMPLEMENTATION {
    STRUCTURE {
      GOBORDER 56@40 AND 216@176
      ENDPORTS {
        INTERFACE controlUserProtocol;
        INTERFACE controlHeatingProtocol;
        INTERFACE toasterSystemControlProtocol;
        }
      }
```

The description of the behavior starts with the determination of the language (C++) and the definition of the generic trigger. In this case a SAP called T of type Timing is used.

```
    BEHAVIOR {
      LANGUAGE 'C++'
      SAPS {
        DEFINE T
          ISA Timing;
        }
```

Next the FSMs (Finite State Machines) are defined. The 'maximum traceability' algorithm keeps the structure of the MSCs and therefore defines a high-level FSM (in ROOM called ROOMcharts) and a set of basic FSMs inside the states of the high-level FSM.

The linear form defines the transitions of the high-level FSM first. The only action that occurs is to set the periodic timer T to 1. This will result in a signal T sent every simulation step.

```
      FSM
        GOBORDER 56@40 AND 328@552
        TRANSITIONS {
          DEFINE initialize
            FROM INITIALPOINT
            TO STATE Idle TRANSITION initialize
              GOFSMPOINT 141@173 AND 147@179 ANGLE 0.0
            GOTRANSITION 141@153 AND 148@177
              LABEL 138@132 AND 152@150 OFFSET 1@-24
              FROM 144@155 TO 144@173
            ACTION 'T.informEvery(1)';
          DEFINE IdSt
            FROM STATE Idle TRANSITION IdSt STATE S1
              GOFSMPOINT 157@237 AND 163@203 ANGLE 0.0
            TO STATE Start TRANSITION IdSt
              GOFSMPOINT 157@261 AND 163@227 ANGLE 0.0
```

```
    GOTRANSITION 157@241 AND 164@225
      LABEL 154@220 AND 168@198 OFFSET 1@-24
      FROM 144@155 TO 144@173;
  DEFINE ToEj
    ...
  }
```

Next all states of the high-level ROOMchart are defined. As they all include a basic ROOMchart, this is defined as well. The order of the definition is again transitions first and substates next but this time without further ROOMcharts inside .

The definition of `Error` is chosen as it shows how to define action with the send function of the protocol and the definition of the trigger event either using a message of a protocol or the SAP `T`.

```
SUBSTATES {
  DEFINE Idle
  ...
  DEFINE Toast
  ...
  DEFINE Error
    GOSTATE 112@264 AND 272@328
      LABEL 0@0 AND 0@0
        GOBORDER 56@40 AND 264@288
        TRANSITIONS {
          DEFINE StEr
            FROM BORDER TRANSITION StEr STATE Start
              GOFSMPOINT 141@149 AND 147@155 ANGLE 0.0
            TO STATE S1
              GOFSMPOINT 141@173 AND 147@179 ANGLE 0.0
            GOTRANSITION 141@153 AND 148@177
              LABEL 138@132 AND 152@150 OFFSET 1@-24
              FROM 144@155 TO 144@173
            ACTION 'controlUserProtocol.send(Empty)';
          DEFINE S1S2
            FROM STATE S1
              GOFSMPOINT 157@237 AND 163@203 ANGLE 0.0
            TO STATE S2
              ...
            TRIGGERS {
              DEFINE SIGNALS {Reset} ON {controlUserProtocol};
              };
          DEFINE ErId
            FROM STATE S2
              GOFSMPOINT 173@325 AND 179@251 ANGLE 0.0
              ...
            TRIGGERS {
              DEFINE SIGNALS {timeout} ON {T};
              };
          }
        SUBSTATES {
          DEFINE S1
```

```
                GOSTATE 112@0 AND 208@64
                  LABEL 0@0 AND 0@0
                GOBORDER 56@40 AND 664@504;
              DEFINE S2
                GOSTATE 112@88 AND 208@152
                  LABEL 0@0 AND 0@0
                GOBORDER 56@40 AND 664@504;
              };
      DEFINE Eject
      ...
      DEFINE Start
      ...
      }
  }
};
```

C.2.4 ControlUserProtocol.port

Now one of the protocol files between actors will be shown. The incoming and outgoing messages are listed in a self-explaining manner.

```
VERSION '1.1'

PROTOCOL CLASS ControlUserProtocol
  SERVICE BasicCommunication
  IN MESSAGES {
    DEFINE Reset ISA Null;
    DEFINE Ready ISA Null;
    DEFINE Start ISA Null;
    }
  OUT MESSAGES {
    DEFINE Comreq ISA Null;
    DEFINE Toast_ack ISA Null;
    DEFINE Empty ISA Null;
    DEFINE Start_ack ISA Null;
    };
```

C.2.5 ToasterSystemHeatingProtocol.port

This file is one of the protocol files between the system actor and an actor. The algorithm has taken the transition label `StEr` and create a signal `Doster` that can be injected when a decision has to be made at that point.

```
VERSION '1.1'

PROTOCOL CLASS ToasterSystemHeatingProtocol
  SERVICE BasicCommunication
  IN MESSAGES {
    }
  OUT MESSAGES {
```

```
DEFINE Doster ISA Null;
};
```

The description of the linear form does not include a more formalized form such as the BNF to describe the syntax. This is due to a lack of documentation of the linear form. The parser source code, however, includes the rules for lexical analyzer `lex`.

D Synthesis log file

This is the log-file created during the 'maximum traceability' synthesis of the toaster model. Comments are added to explain what the algorithm is doing.

First the structure is created. That is the actors, the messages in all bMSCs and finally the protocols between actors.

```
A) hMSC ROOMstructure ...

 - create package Toaster
 - create system actor ToasterSystem
 + found hMSC node: IDLE
 - create actor: USER
 - create actor: CONTROL
 - create actor: HEATING
 - create protocol: ControlUserProtocol
 + found signal: COMREQ
 - create protocol: ControlHeatingProtocol
 + found signal: COOL
 + found hMSC node: TOAST
 + found signal: TOAST_ACK
 + found signal: WARM
 + found hMSC node: ERROR
 + found signal: EMPTY
 + found signal: RESET
 + found signal: ERROR
 + found hMSC node: EJECT
 + found signal: READY
 + found signal: EJECT
 + found signal: EJECT_DONE
 + found hMSC node: START
 + found signal: START
 + found signal: START_ACK
 + found signal: HOT
```

Next the behavior is created. For every actor a high-level ROOMchart is built including the states and all the transitions. As this is exactly the the same process for all actors, it is only shown here for the actor USER.

```
B) ROOMbehavior (algorithm maximum traceability) ...

* creating actor USER
```

```
 - create state Idle in the hROOMchart
 - create state Toast in the hROOMchart
 - create state Error in the hROOMchart
 - create state Eject in the hROOMchart
 - create state Start in the hROOMchart
 - create transition initialize from INITIALPOINT to Idle
 - create outpoint initialize for chart INITIALPOINT in actor USER
 - create inpoint initialize for chart Idle
 - create transition IdSt from Idle to Start
   ...

* creating actor CONTROL

 - create state Idle in the hROOMchart
   ...
 - create transition initialize from INITIALPOINT to Idle
 - create outpoint initialize for chart INITIALPOINT in actor CONTROL
 - create inpoint initialize for chart Idle
   ...

* creating actor HEATING

 - create state Idle in the hROOMchart
   ...
 - create transition initialize from INITIALPOINT to Idle
 - create outpoint initialize for chart INITIALPOINT in actor HEATING
 - create inpoint initialize for chart Idle
   ...
```

Next the message traces are built for every bMSC of every actor. The trace has the following format (source state number, received message, send message, destination state number). At that point, no optimization is done. Again, only parts of the output is shown.

```
* in actor USER

 - goto bMSC Idle
  + found message COMREQ
  - create trace (0, COMREQ, NULL,1)
  - number of states is 1
  - create state(s) 1
  - read trace (0,1)
  - create transition from inpoint initialize to state 1
  - create transition from inpoint ErId to state 1
  - create transition from inpoint EjId to state 1
  - create transition from state 1 to outpoint IdSt

 - goto bMSC Toast
  + found message TOAST_ACK
  + found message COMREQ
```

```
  - create trace (0, TOAST_ACK, NULL,1)
  - create trace (1, COMREQ, NULL,2)
  - number of states is 2
  - create state(s) 1, 2
  - read trace (0,1)
  - create transition from inpoint StTo to state 1
  - read trace (1,2)
  - create transition from state 1 to 2
  - create transition from state 2 to outpoint ToEj
  - create transition from state 2 to outpoint ToSt

 - goto bMSC Error
  + found message EMPTY
  + found message RESET
  - create trace (0, EMPTY, NULL,1)
  - create trace (0, NULL, RESET, 1)
  - number of states is 1
  - create state(s) 1
  - read trace (0,1)
  - create transition from inpoint StEr to state 1
  - create transition from state 1 to outpoint ErId

 - goto bMSC Eject
  + found message READY
  - create trace (0, NULL, READY, 1)
  - number of states is 1
  - create state(s) 1
  - read trace (0,1)
  - create transition from inpoint ToEj to state 1
  - create transition from state 1 to outpoint EjId

 - goto bMSC Start
  + found message START
  + found message START_ACK
  - create trace (0, NULL, START, 1)
  - create trace (1, START_ACK, NULL,2)
  - number of states is 2
  - create state(s) 1, 2
  - read trace (0,1)
  - create transition from inpoint IdSt to state 1
  - create transition from inpoint ToSt to state 1
  - read trace (1,2)
  - create transition from state 1 to 2
  - create transition from state 2 to outpoint StEr
  - create transition from state 2 to outpoint StTo

* in actor CONTROL

 - goto bMSC Idle
```

```
   ...

* in actor HEATING

 - goto bMSC Idle
   ...
```

As a requirement of ROOM, the first trigger event of every basic ROOMchart has to moved to the last transition of its predecessor.

If a transition originates in the initial point and has a trigger event, then an additional state is created and the transition is split up into two transitions. This is a work-around the problem that ROOM does not allow a trigger event for the initial transition.

```
* in actor USER
 - create extra state in Idle coming from INITIALPOINT
 - moved trigger from Start to Idle
 - moved trigger from Eject to Toast
 - moved trigger from Start to Toast
 - moved trigger from Idle to Error
 - moved trigger from Idle to Eject
 - moved trigger from Error to Start
 - moved trigger from Toast to Start

* in actor CONTROL
 - moved trigger from Start to Idle
 - moved trigger from Eject to Toast
 - moved trigger from Start to Toast
 - moved trigger from Idle to Error
 - moved trigger from Idle to Eject
 - moved trigger from Error to Start
 - moved trigger from Toast to Start

* in actor HEATING
 - create extra state in Idle coming from INITIALPOINT
 - moved trigger from Start to Idle
 - moved trigger from Eject to Toast
 - moved trigger from Start to Toast
 - moved trigger from Idle to Error
 - moved trigger from Idle to Eject
 - moved trigger from Error to Start
 - moved trigger from Toast to Start
```

The following step is used to detect whether a decision is made at some point. If this is the case, a protocol plus the necessary signals is created to allow the use of injection points during simulation.

```
* in actor USER
 ? no injection point test in INITIALPOINT
 + choice found after state Toast
  + found successor Eject
  + trans ToEj has sap T -> delete SAP
```

```
 - create protocol: ToasterSystemUserProtocol
 + create signal DoToEj
  - add trigger DoToEj to transition ToEj
  + found successor Start
  + trans ToSt has sap T -> delete SAP
 + create signal DoToSt
  - add trigger DoToSt to transition ToSt
 + choice found after state Start
  + found successor Error
  + trans StEr has trigger Empty -> no injection needed
  + found successor Toast
  + trans StTo has trigger Toast_ack -> no injection needed

* in actor CONTROL
 ? no injection point test in INITIALPOINT
 + choice found after state Toast
  + found successor Eject
  + trans ToEj has trigger Ready -> no injection needed
  + found successor Start
  + trans ToSt has trigger Start -> no injection needed
 + choice found after state Start
  + found successor Error
  + trans StEr has trigger Error -> no injection needed
  + found successor Toast
  + trans StTo has sap T -> delete SAP
 - create protocol: ToasterSystemControlProtocol
 + create signal DoStTo
  - add trigger DoStTo to transition StTo

* in actor HEATING
 ? no injection point test in INITIALPOINT
 + choice found after state Toast
  + found successor Eject
  + trans ToEj has trigger Eject -> no injection needed
  + found successor Start
  + trans ToSt has trigger Hot -> no injection needed
 + choice found after state Start
  + found successor Error
  + trans StEr has sap T -> delete SAP
 - create protocol: ToasterSystemHeatingProtocol
 + create signal DoStEr
  - add trigger DoStEr to transition StEr
  + found successor Toast
  + trans StTo has trigger Warm -> no injection needed
```

As a last step all files are written. The algorithm uses internal data structures to keep track of all the ROOM model objects and names. Now this information is written to files.

```
- Writing file ./Toaster.package ...
```

```
- Writing file ./ToasterSystem.actor ...
- Writing file ./ControlUserProtocol.port ...
- Writing file ./ControlHeatingProtocol.port ...
- Writing file ./ToasterSystemUserProtocol.port ...
- Writing file ./ToasterSystemControlProtocol.port ...
- Writing file ./ToasterSystemHeatingProtocol.port ...
- Writing file ./User.actor ...
- Writing file ./Control.actor ...
- Writing file ./Heating.actor ...
```