Executable Model Synthesis
and
Property Validation
for
Message Sequence Chart Specifications

by

Piotr Konrad Tysowski

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical & Computer Engineering

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Message sequence charts (MSC's) are a formal language for the specification of scenarios in concurrent real-time systems. The thesis addresses the synthesis of executable object-oriented design-time models from MSC specifications. The synthesis integrates with the software development process, its purpose being to automatically create working prototypes from specifications without error and create executable models on which properties may be validated. The usefulness of existing algorithms for the synthesis of ROOM (Real-Time Object Oriented Modeling) models from MSC's has been evaluated from the perspective of an applications programmer according to various criteria. A number of new synthesis features have been proposed to address them, and applied to a telephony call management system for illustration. These include the specification and construction of hierarchical structure and behavior of ROOM actors, views, multiple containment, replication, resolution of non-determinism and automatic coordination. Generalizations and algorithms have been provided. The hierarchical actor structure, replication, FSM merging, and global coordinator algorithms have been implemented in the Mesa CASE tool. A comparison is made to other specification and modeling languages and their synthesis, such as SDL, LSC's, and statecharts. Another application of synthesis is to generate a model with support for the automated validation of safety and liveness properties. The Mobility Management services of the GSM digital mobile telecommunications system were specified in MSC's. A Promela model of the system was then synthesized. A number of optimizations have been proposed to reduce the complexity of the model in order to successfully perform a validation of it. Properties of the system were encoded in Linear Temporal Logic, and the Promela model was used to automatically validate a number of identified properties using the model checker Spin. A ROOM model was then synthesized from the validated MSC specification using the proposed refinement features.

Acknowledgments

I sincerely wish to thank my supervisor, Prof. Dr. Stefan Leue, at the Institute for Informatics of the Albert-Ludwigs-Universität in Freiburg, Germany, for his guidance, expertise, and patience.

I also wish to thank Mohammad Zulkernine at the University of Waterloo for his contribution to an early version of a manuscript that he and I co-authored on the specification and validation of a hand-coded variant of the GSM model that motivated the Promela model synthesis work described in this thesis.

Lastly, I wish to thank my two additional readers at the University of Waterloo, Prof. Kontogiannis and Prof. Seviora.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Software engineering

Empirical studies have shown that many software faults occur due to errors introduced in the requirements and specification phase of software construction[1]. This is of concern in real-time telecommunication systems, which are highly complex in nature and consist of concurrent, communicating processes. The accurate specification of protocols and services, and the verification of fulfillment of these services, are needed to reduce the time and cost of development.

Software engineering is the discipline of building reliable, cost-effective software solutions to practical technological problems, usually involving large and complex systems, by applying scientific knowledge and techniques [4]. The practitioner must follow a precise methodology to create a working software system that operates according to user requirements upon completion, avoiding unnecessary delays and cost overruns in the process. The steps to follow are outlined in a process model, of which there are several types according to the characteristics of the project, especially the extent to

[1] One study, by DeMarco [29], found that 56% of all bugs detected can be traced to errors made during the requirements stage.

which risk analysis and client involvement plays a role in the requirements [1]. For example, the traditional waterfall model[2] in figure 1.1, has been found to be overly documentation-centric and involving the customer to an insufficient extent, resulting in products not closely meeting his expectations. More recent derivatives such as the spiral model, depicted in figure 1.2, have been considered more appropriate for large projects of higher risk, as they incorporate risk analysis at every stage of the development process.

1.2 Requirements specification

The functional and non-functional[3] requirements[4] of the software system are first captured through elicitation from the various parties to be involved with the eventual product, including the customers and actual users. An analysis is then undertaken to identify possible inconsistencies and omissions. The updated requirements are next codified in a specification document, either in natural language, or, preferably, one that uses a standardized language with formal semantics. Finally, the requirements are validated with the customer as a final assurance of their correctness and completeness through a comprehensive test suite[5].

The requirements are verified for consistency and completeness using formal techniques, with the aid of automated analysis tools whenever possible.

[2] An iterative variant exists, in which feedback is obtained from the operations phase, and any necessary changes to requirements result in a new iteration of development.

[3] Non-functional requirements include performance constraints and user interfaces.

[4] The IEEE Standard 729 [28] defines a *requirement* as a "condition or capability that must be met or possessed by a system ... to satisfy a contract, standard, specification, or other formally imposed document."

[5] The terms *verification* and *validation* applied in the software engineering context are treated here as having distinct meaning. *Verification* refers to a formal (i. e. mathematical) proof of correctness of a software system model with respect to a set of required properties, using theorem-proving techniques, for instance. *Validation,* on the other hand, is the process of establishing correctness with respect to a set of properties or identifying errors through experimentation, including testing and model-checking through state space exploration of a finite state model.

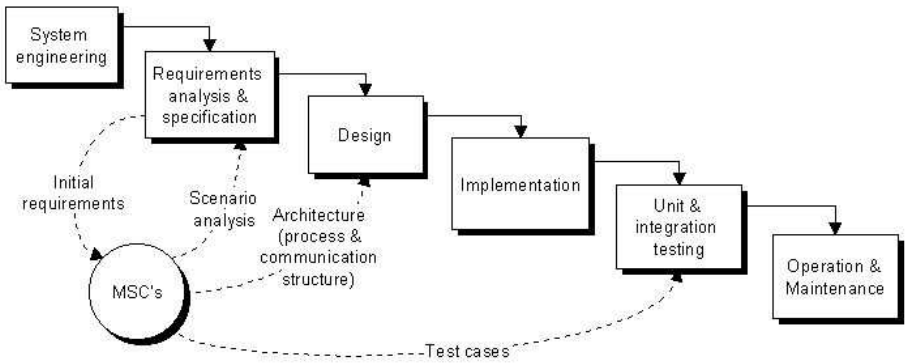

Figure 1.1: The waterfall process model [1]. The role of MSC's in this model, indicated by the dashed interaction lines, is explained in Section 2.1.

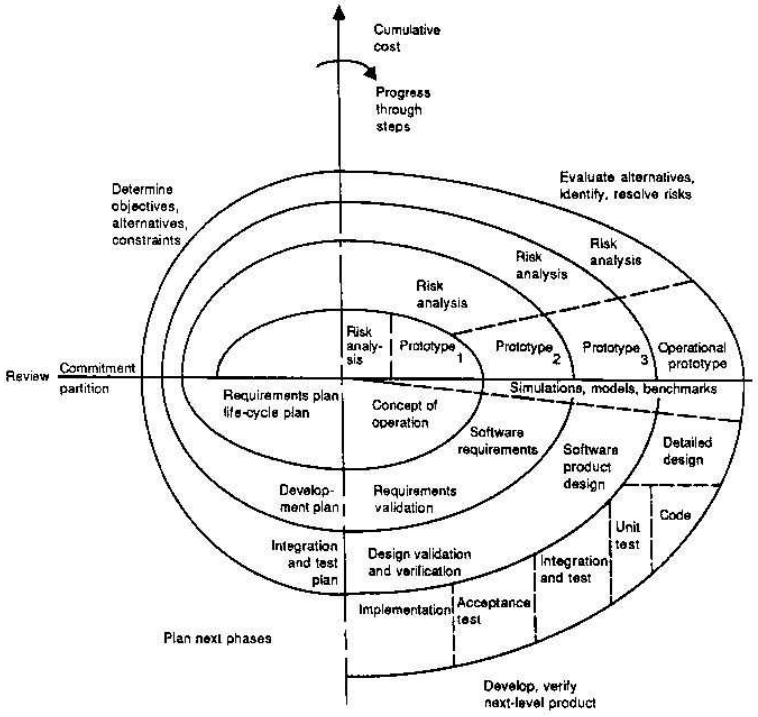

Figure 1.2: The spiral process model [1]. ©Prentice-Hall, 1991.

Once the requirements have been verified and understood, a design is drawn to capture an architectural view of the system that illustrates, on a relatively high level of abstraction, its structure and behaviour. Language- and machine-specific constructs and syntax are avoided. The structural component of the design consists of a mapping of the desired properties of the system onto physically-realizable structures or executing software processes, while the behaviour modeling is expressed in terms of the interaction of the structures through signals [6]. The description of the interrelationships between processes in terms of temporal progress through scenarios is called *inter-object specification* [14]. An external environmental structure may be included to act in the capacity of a user, assigning tasks and accepting output. In general, the design presents a framework for the implementation of the software system.

1.3 Visual formalisms and scenario-based design

A number of requirements engineering formalisms with visual components have been standardized. Visual notation is often easier for humans to analyze, especially non-experts.

1.3.1 UML

UML (Unified Modeling Language) is a standard modeling language for the specification, visualization, construction, and documentation of the artifacts of software systems [32]. Behavior can be specified using sequence interaction diagrams, based on those of Booch [45]. Collaboration diagrams that specify object interaction can also be used. UML 1.1 was produced in 1997 by the UML Partners consortium, including DEC, HP, IBM, Microsoft, Oracle, ObjecTime, Rational, and others, in response to an RFP (Request for Proposal) issued by the OMG (Object Management Group), which has

[6] The terms *signal* and *message* are used interchangeably in the text. Any data content attached in a signal is irrelevant to the discussion herein.

approved it as a standard.

Sequence diagrams are used to specify the dynamic structure of a system, or how it behaves, rather than how it is structured [41, 42, 43]. A sequence diagram shows the interaction of objects in terms of the messages that they exchange arranged in time sequence [33]. Sequence diagrams resemble MSC's, in which the vertical dimension represents time, while objects are arbitrarily arranged horizontally. The duration of existence of an object is represented by a vertical, dashed *lifeline*. An activation is a period during which a process is active and performing an operation, and is represented by a thin rectangle. The activation represents the focus of control in the system. A process activation is triggered by the receipt of a message and comprises the subsequent processing that needs to take place. An example of a UML sequence diagram appears in figure 1.3.

Variations on the symbology are possible, if required. For instance, an asynchronous message can be drawn with a half-arrowhead, and activations can overlap in time. An asynchronous message can create a new thread, create a new object, or communicate with a thread that is already running. A (synchronous) procedure call can be drawn as a full arrowhead, with a return shown as a dashed arrow, and activations cannot overlap in time. Branches are indicated by multiple arrows leaving a single point, each labeled by a guard condition. The branch represents conditionality if the guard conditions on all the messages are mutually exclusive, in which case only one message is sent. On the other hand, the branch represents concurrency if the guard conditions on the messages are mutually inclusive, so that multiple messages are sent. It is also possible to show features such as process creation, destruction (indicated by an **x**), and recursion. Iteration, in which a message is sent multiple times to multiple receiver objects, is indicated by an asterisk followed by square brackets containing an iteration expression specifying the number of iterations.

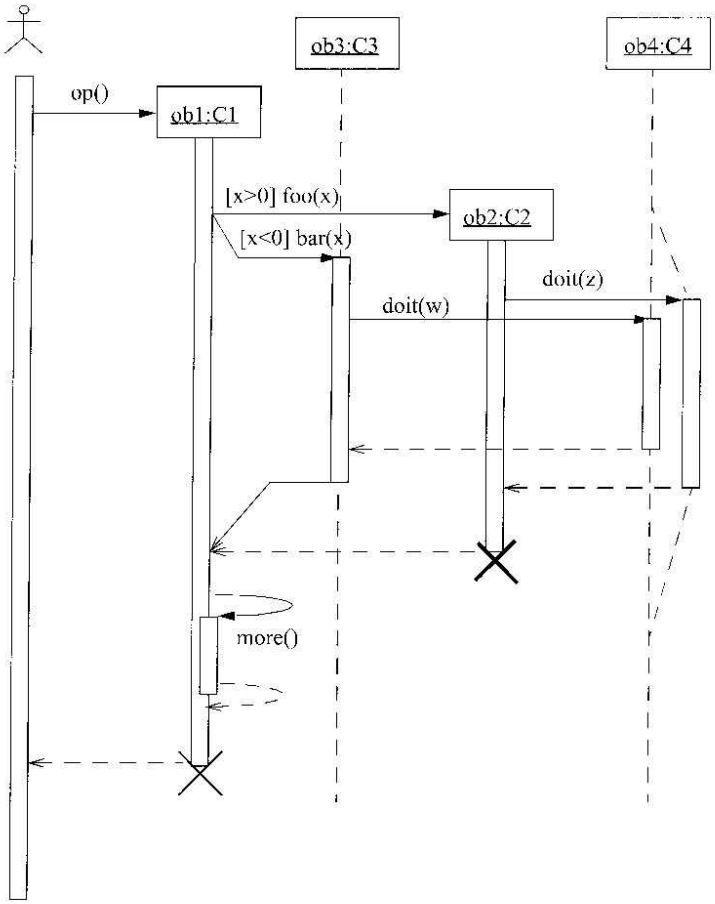

Figure 1.3: A UML sequence diagram [33]. The $[x > 0]$ and $[x < 0]$ conditions are guards. The result of the $[x > 0]$ operation results in the creation of the `ob2` class role object. The `doit()` labels specify operations. The `ob4` role splits into two side-by-side lifelines, each track being a conditional branch, depending on which message from `ob3` or `ob2` is received. The `more()` procedure is executed in a nested thread, and is termed a *self-call*. The `ob1` and `ob2` class role objects are destroyed at the end of the sequence.

1.3.2 FSM's

FSM's (Finite State Machines), also known as DFA's (Deterministic Finite Automata) can be used to specify control in systems that are in one of a finite set of possible states, and where state transitions are triggered by events. They are the underlying automaton for many formal languages and models. They consist of:

- A finite set of states, Q.
- A finite set of inputs, I.
- A finite set of outputs, O.
- A transition function $f : Q \times I \to Q \times O$ where f is a total or partial function.

FSM's are a synchronous model in that the system must be in one global state at any time, and only a single transition can occur. Thus, a basic FSM cannot model a concurrent and asynchronous system, as can an MSC.

Two equivalent models for state machines exist:

Mealy. The outputs are a function of both the present state and the input.

Moore. The outputs are associated with the states of the system.

1.3.3 OMT

OMT (Object Modeling Technique) is a modeling methodology with rich notation that addresses object-oriented concepts such as classification, polymorphism, and inheritance. The technique is divided into three parts: analysis, system design, and object design. The latter is the refinement of object structure towards efficient implementation. The system is described in aggregate from three different views:

Object model. Utilizes object diagrams to describe the structure of the objects in the system and their interrelationships.

Functional model. Utilizes data flow diagrams to describe process computations.

Dynamic model. Utilizes state diagrams for every class to describe the control flow in the system. Event flow diagrams are first drawn to identify events, states, and transitions.

A state diagram consists of entry/exit actions, guarded transitions, actions (instantaneous operations performed on transitions), output events on transitions, and internal actions (where an event does not cause a state to change). State diagrams provide aggregation and generalization using the concept of inheritance. States can be decomposed in sub-state diagrams, where the transitions of the super-state are inherited by each of its sub-states. An event hierarchy is also provided to generalize or specialize events. Sub-events inherit attributes from super-events. The state diagram of a subclass is usually an independent orthogonal concurrent addition to the state diagram.

1.4 Real-time systems

Rapid advances in the telecommunications industry have precipitated an increased interest in distributed, reactive real-time systems. These maintain an ongoing interaction with the environment by responding to stimuli and operating in an event-driven fashion. Tasks are executed in parallel for greater efficiency and throughput, and protocols are defined for the communication necessary to co-ordinate all activities in the system. Processing is done by interacting concurrent processes or threads. Real-time systems are characterized by their stringent timing requirements to satisfy specified deadlines for every task. The constraints consist of the *service time* for a request and the *latency*, the delay before processing begins on the input.

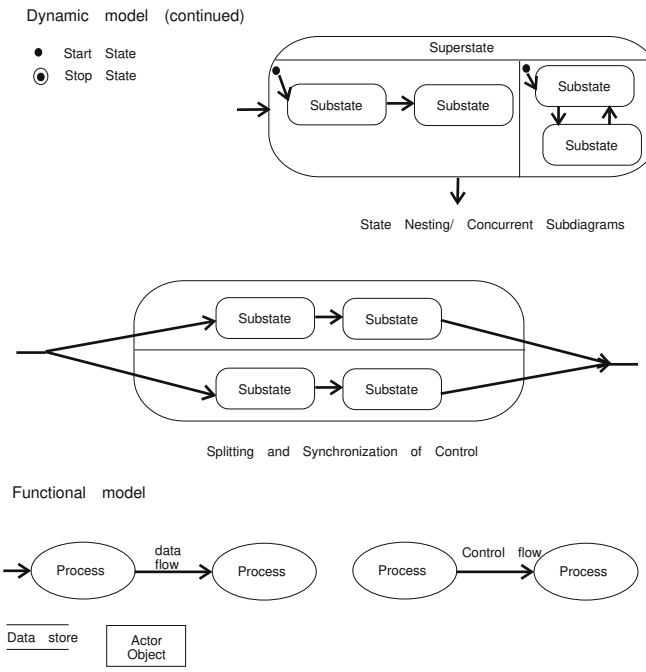

Figure 1.4: An example of an OMT dynamic and functional model [34]. State diagrams can be nested. The states in the nested diagram are all refinements of the state in the high-level diagram. The states in a nested diagram can interact with other states, even outside of the diagram. Actions can be associated with the entry and exit of a state. Concurrency is modeled by an object split into multiple sub-diagrams. The forked transitions into and merged transitions out of the sub-diagrams entail synchronization of activities. The merging of concurrent control occurs when the target state becomes active due to all transition conditions being satisfied, in any order. The functional model can show the flow of data between objects. An *actor* is an active object that produces or consumes data values, acting as a source or a sink. A *data store*, on the other hand, is a passive object that stores data for later access [44].

Reactive systems do not terminate under normal circumstances; they continuously respond to stimuli from the environment. Real-time systems are frequently found in embedded technologies. Examples of real-time applications are: operating systems, communication devices and their protocols, and robotics.

Transformational systems, on the other hand, are different in that they transform a set of input data into a set of output data in a finite computation, then terminate. No constraint is placed on the response or execution time, and no interaction with the environment is present. Examples of transformational applications are: numeric computations in engineering, compilers, and financial bookkeeping.

Real-time systems often use *asynchronous* communication, in which the sender of a signal continues its execution immediately after the signal is dispatched, rather than blocking and waiting for a reply. Thus, the send operation is *non-blocking,* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer and the transmission of the message proceeds in parallel with the sending process [30]. Each process maintains a buffered message input queue from which it retrieves its incoming requests. The receive operation can be of a *blocking* or *non-blocking* variant. In the *non-blocking* variant, the receiving process proceeds with its program after issuing a receive operation, which causes a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled through polling or an interrupt. The *blocking* variant, assumed in this thesis, is simpler in that a process blocks until a message arrives. Although *non-blocking* communication can be more efficient, it does involve extra complexity in the receiving process due to the need for acquiring an incoming message out of its flow of control.

The advantage of asynchronous communication is that the interval during which the behavior of a process is unresponsive is short, and the communication mechanism required to support it is relatively simple, making it suitable in situations where a rapid response is critical [3]. In *synchronous* communication, the sending process is blocked until it receives a reply, and

during this interval, it cannot respond to any other inputs. Although this increases the control over ordering of activities, it can be less efficient than asynchronous communication.

One example of a major distributed telecommunications system using the real-time approach is that of the mobility management service of the GSM (Global System for Mobile Communications) cellular phone system in Europe, where base stations co-ordinate seamless hand-offs of calls due to the physical relocation of phones [2].

1.5 Synthesis

Synthesis, in this context, is the transformation from a requirements specification to an architectural design, from one language to another. One form of synthesis comprises the problem of computationally constructing a structurally and behaviorally equivalent (or determinable) state-based model from scenarios[7], a decidedly non-trivial task given the rich syntax and semantics of present-day specification languages. Both the source and destination models are formalized, and the translation is often complex and tedious even when applying it to software systems of low complexity. It is desirable to utilize a CASE (Computer-Aided Software Engineering) tool to automatically generate a model from a specification that is useful to a programmer for purposes of refinement and testing, using well-defined translation rules. Such a productivity tool can be essential to the increasingly common rapid development project[8] [7].

The purpose of synthesis is two-fold:

1. To create a design consistent with the requirements for use as a prototype or as a framework for the implementation.

[7]Scenarios describe the possible event sequences that the system may execute.

[8]Rapid development emphasizes development speed using schedule-oriented practices. CASE tools, with sufficient training and experience, can decrease the time and risks associated with the transition to the design phase.

2. Translate the requirements to an executable model for the purpose of verifying whether it satisfies functional requirements.

In this thesis, Section 4 addresses the first area, while Section 5 investigates the second. A process model incorporating synthesis is suggested in Section 5.6.

1.6 Scope & methodology

1.6.1 ROOM synthesis

The underlying methodology of a synthesis process requires a well-defined approach, terminology capturing conceptual constructs, a suitable notation describing the syntax and semantics of the modeling language, and a process being prescribed, including activities, outputs, and any heuristics [18]. The synthesis process will thus be described.

A popular way of describing message interaction between parallel processes is through an MSC (Message Sequence Chart) specification. This language, comprised of a pictorial and equivalent textual form, was standardized in ITU-T (International Telecommunications Union) Recommendation Z.120. MSC's specify scenarios consisting of message-passing activities between system components. Components are instances of software processes, and are represented by vertical lines, while message transmissions are denoted by directed arrows joining the process constructs along a vertical time axis, as shown in Fig. 1.5. MSC's are useful for concisely specifying *scenarios* in communication protocols. A scenario is an ordered sequence of actions relating to external and internal requirements. Each MSC represents a possible execution trace, and it is possible to combine multiple charts into a cohesive system description containing all possible execution paths, although some limitations to expressivity exist, as discussed later. A more in-depth discussion of MSC's is found in Section 2.1, and a comparison with a related state machine formalism called SDL is covered in Section 3.1.

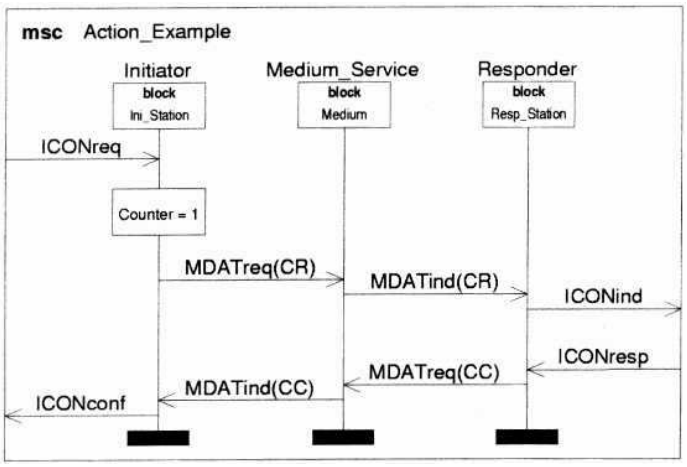

Figure 1.5: An example of a bMSC (basic MSC), defining signal transmissions between three process instances and an environment [8].

Although there exist executable semantics for MSC specifications that rely on universal assumptions, [9] such as SDL [25] (discussed in Section 3.1), a useful byproduct of software development is an early working prototype with which to test functional and approximated performance requirements, and to demonstrate progress to customers for their feedback. Also, it is beneficial to advance to the design stage and create a realizable architecture using tool automation to minimize errors and save time.

At least two different views of the continuity of the design process exist:

Continuous. A requirements specification model is refined into a design model. Although this process can save time, minimize errors, and result in high traceability, the design will closely resemble the requirements model, even though the two can theoretically be built quite differently. The requirements model may need to anticipate and conform to design-related parameters. For example, a design synthesized from

[9] The system must always follow a specified behavior whenever a set of specific conditions is satisfied, regardless of the history. Universal conditions are given more treatment in the discussion on LSC's in Section 3.3.

an MSC specification may assume an asynchronous system requiring the presence of a context-switching process model and independent message buffers.

Discontinuous. A requirements specification model is discarded before the design model is built from scratch. This approach may result in a design that is better optimized for a target computer system, but is more time-consuming, resulting in a design that can be difficult to verify, and be more prone to errors. For instance, a synchronous protocol specified in an MSC may be built most efficiently by adopting a one-process layered design using function invocation and global variables for inter-layer communication, rather than message-passing and expensive context-switching. However, functional testing would need to be relied on for validation, rather than model-checking of the requirements.

The application of the continuous design process is advocated in this thesis, because automation of the software development process is one of the goals of the work. A design-time model can be synthesized automatically from an MSC specification. This thesis is based on the process of synthesizing design models in the ROOM (Real-Time Object-Oriented Modeling Chart) [3] language, also known as ROOMcharts[10], first investigated by Leue, Mehrmann, and Rezai [5, 6]. ROOM models are a useful technique in capturing software designs of concrete software engineering problems, as a step in the software process model following the formal requirements phase. ROOM allows the expression of domain knowledge and design ideas, with emphasis on the ability to describe real-time systems. It can capture high-level system properties, such as communication ports and process structures, in addition to the low-level abstractions inherent in procedural programming languages, such as variables and timers. ROOM supports an event-based asynchronous communication model, so that it is appropriate for describing communication protocols specified in MSC's.

[10]The terms ROOM and ROOMcharts are used interchangeably in the text.

"ObjecTime Developer," henceforth referred to as OTD, is a software development suite produced by ObjecTime Ltd. in Canada. It permits a software engineer to enter designs consistent with the requirements, refine them, and build executable models to run animated simulations. A sample session is shown in Figure 1.6. OTD is a commercial-grade CASE tool, first developed in the late 1980's, and successfully utilized in a number of large industrial projects, including the OAM (Operations, Administration, and Maintenance) software of Nortel Networks's 10 Gb/s S/DMS OC-192 TransportNode network element [31].

The synthesis process accepts as its input a complete MSC specification, entered into another CASE tool called MESA (Message Sequence Chart Editor, Simulator and Analyzer) [21] [11]. This tool enables the user to compose a system model conforming to the MSC standard, and automatically run syntactic analysis on it to detect any errors in the construction. Potential problems such as timing inconsistencies, non-local choice, and process divergence are automatically detected through static model-based analysis. A sample open project MESA session is shown in figure 1.7.

One of the reasons for standardizing MSC's was to support the creation of such tools. Synthesis algorithms have already been implemented in MESA that translate MSC specifications into ROOM models. The synthesis involves a translation from an MSC specification graph stored in memory to a so-called *linear form* file that contains a complete description of the generated ROOM model. This file is then imported into OTD. An example of a linear form file appears in Appendix A.

Although the theoretical groundwork for this type of synthesis has already been laid out, there is ample room to refine the generation to produce a design-time model that is more practical and useful to a typical developer. One of the goals of the research is to create accessible and practical tools for

[11] MESA is available under a research license from Prof. Stefan Leue of the Albert-Ludwigs-Universität in Germany. The SunOS 5.x and Linux i386 2.2.x platforms are supported. MESA is a C++ application with Tcl/Tk GUI components, and a Tcl/Tk 8.x installation is required. For more information, visit http://fee.uwaterloo.ca/~mesa

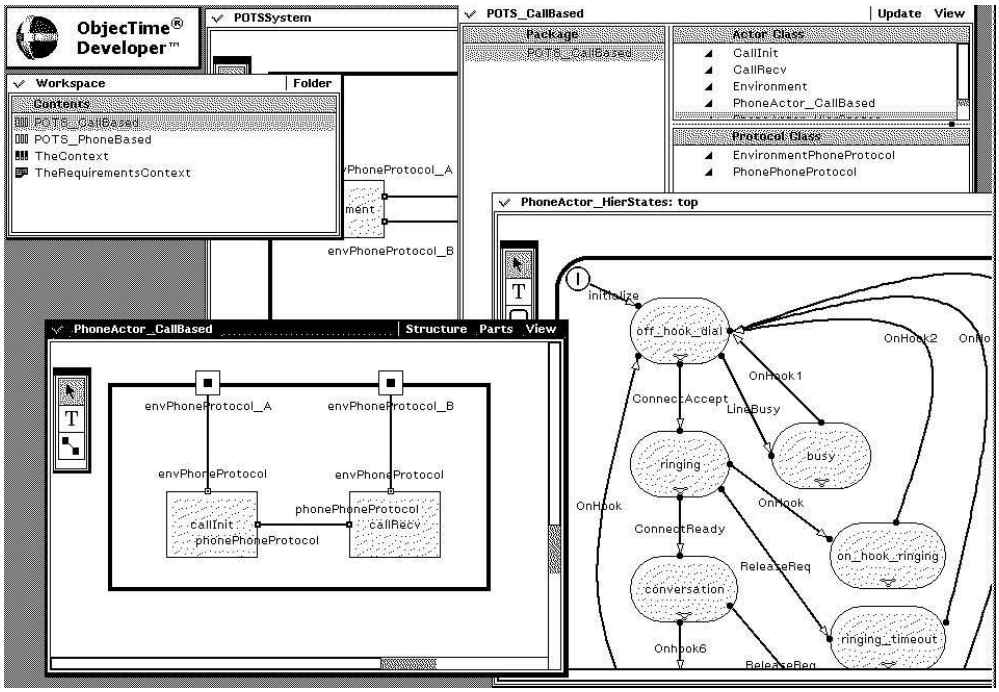

Figure 1.6: A sample screen-shot of an ObjecTime Developer session, with the hand-coded PBX project being displayed (discussed later, in Section 4.1.1).

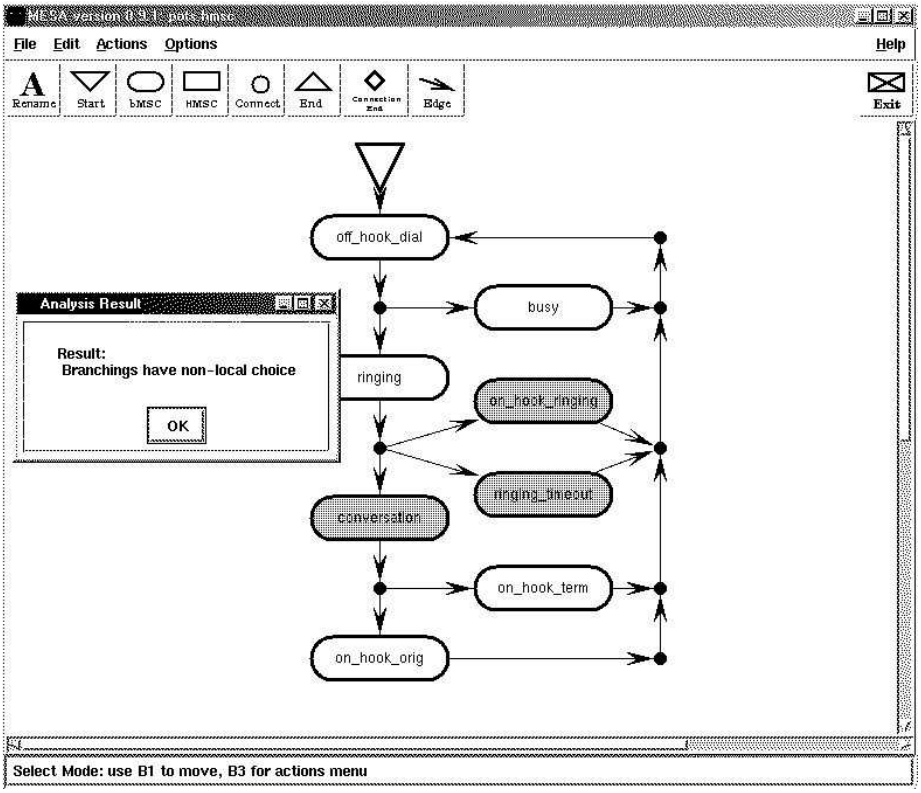

Figure 1.7: A sample screen-shot of a MESA session, with static analysis being run on the PBX specification. Non-local branches have been discovered in the highlighted bMSC's. Non-local choice is discussed later in Section 4.5.5.

solving real problems in the engineering field. Hence, an analysis has been undertaken of the synthesis results produced so far in an effort to identify areas of improvement for further study. The process involved composing an MSC specification conforming to the ITU-T Z.120 standard using MESA, formally recording the requirements of a sample system of sufficient complexity to observe any shortcomings in the end product of the synthesis, including structural layout, connectivity, branching, etc. A ROOM design model was synthesized from the MSC specification by application of synthesis algorithms. The result was compared to a ROOM model of the same system entered from scratch by hand, using the facilities provided within OTD.

Automatic synthesis confers a number of advantages onto a software developer:

- Shorter development time through easy-to-use interfaces and automation, and, as a result, lower cost.
- Fewer or no errors in the translation of the specification to a high-level design.
- Documentation for the purposes of maintenance and test plan creation.
- Easy modification through automatic re-generation of a model when a change, however significant, is made in the specification.
- Demonstrability of a working C++ prototype to a customer for early feedback.

1.6.2 Promela synthesis & validation

Another important application of synthesis is to enable safety and liveness requirements to be validated, something that is not possible with MSC's alone. This requires the specification and validation of the properties in a suitable temporal language such as LTL (Linear Temporal Logic). The

Spin Model Checker is a tool that supports the validation of LTL properties on Promela design models, and so a synthesis feature has been added to MESA to generate Promela code. The system under study in this thesis is the Mobility Management service of the GSM cellular telecommunications system, responsible for call management and handover. MSC's were used to specify the high-level requirements of operational behaviour derived from the GSM standard. The specification was entered and verified using the MESA editor and analyzer tool, and a model in the Promela design language was automatically synthesized. Various safety and liveness properties were encoded in Linear Temporal Logic, and automatically proven to hold true of the Promela model using Spin. The validation process and results obtained are fully discussed.

1.7 Overview

The objective of the thesis is to understand the need and methodology for the synthesis of real-time systems from requirements specifications, and the underlying theory of the languages involved in the process.

1.7.1 Summary of Chapters 2-3

The first chapter presents background on MSC's and ROOMcharts, the source and target of the synthesis work explored in the thesis. The second chapter discusses and compares related work on specification languages and their synthesis, including Live Sequence Charts.

1.7.2 Summary of Chapter 4

Chapter 4 of the thesis concerns the synthesis of ROOM design models to create working design-time prototypes from MSC specifications. The system under study is the call management service of a small telephone exchange. Hand-coded and automatically generated models using the synthesis facili-

ties in the MESA CASE tool were enhanced using proposed new techniques. The content is as follows:

- The communication protocols in today's telecommunication software systems are characterized by high complexity. The need for a rigorous development process, including formal specification, verification, and design synthesis, can prevent errors in the development process, as discussed in Section 1.1.

- A number of specification and design languages are presented in Section 2, and their theoretical basis and support for synthesis are explained. The focus of the thesis is on mapping MSC specifications to ROOMchart designs, and both models, as well as the synthesis process itself, are described in detail.

- In Section 4.1, an evaluation was first undertaken of current synthesis results, and what improvements were necessary to adapt a proof-of-concept result to one with high potential for commercial use. A number of criteria to satisfy were identified.

- The first area investigated was the support for multiple views of a system component, as discussed in Section 4.2.

- The use of replication, as suggested in Section 4.3, permits population of actors and interfaces in the system.

- Next, a construction for a hierarchical structural layout of the system is described in Section 4.4.1. Various options for the specification of hierarchical behavior are described in Section 4.4.2. Generalized algorithms for all ROOM-compliant models are provided. The use of hierarchy aids in organizing the knowledge of the system.

- Non-determinism can exist in a scenario-based model. Various constructions for synchronization and simulation of system processes are suggested in Section 4.5.

- The simulation of ROOM models in Section 4.6 enables implementation-time MSC validation to be performed.

1.7.3 Summary of Chapter 5

The topic of Chapter 5 of the thesis is the synthesis of a model not for the purpose of creating a design for refinement, but to formally capture and automatically validate safety and liveness properties specified in a language called Linear Temporal Logic.

The content of the section is as follows:

- Section 5.2 presents an overview of the GSM system and the Mobility Management services, defining the operational requirements under study. An informal explanation of the handover procedure is also included.
- Next, in Section 5.3, a brief introduction to the Message Sequence Charts specification language and MESA tool is provided, as well as the Promela design language and Spin tool.
- In Section 5.4, the Promela model for GSM that was synthesized from the MSC specification is described in detail.
- In Section 5.5, Linear Temporal Logic (LTL) is explained, and the formulæ for high-level requirements and the validation results are presented, including the memory and processing resources required.
- Finally, in Section 5.6, a brief conclusion and some future research directions are presented.

1.7.4 Summary of Chapter 6

The GSM system of Chapter 5 was synthesized using the features proposed in Chapter 4. Thus, this system was first validated through Promela syn-

thesis and LTL formulæ, then synthesized into a ROOM design-time model, following the suggested development process.

1.7.5 Summary of Chapter 7

The purpose and process of synthesis is summarized with concluding notes and directions for future research.

1.8 Contribution

I have presented the need for the synthesis of real-time software systems from Message Sequence Chart specifications, and researched and explained the underlying theory of various languages and synthesis processes, identifying features of importance and performing a comparison. I have suggested the benefits of synthesis in the software development cycle, and explained its role in design-time prototyping and property validation. I have undertaken an analysis of the existing synthesis techniques and evaluated their usefulness from an application programmer's perspective based on criteria that I have identified.

Next, I have suggested several new improvements to the synthesis of ROOM models, providing examples and generalized inter-compatible algorithms for implementation. The hierarchical actor structure allows for planning of the system layout. Views and replication assist in the composition of multiple processes. The global co-ordination model allows for automatic execution of the model with resolution of non-local choice branches. Improvements to hierarchical state machines in the behaviour of ROOM actors has also been investigated. The advantages conferred by these extension features include the following:

- The addition of structural object-based concepts to the system for improved decomposition and understanding.

- The automation of decision-making in the model for the purpose of hands-free simulation.
- The ability to create multiple views and instances of processes increases the support for more complex scenarios.

I implemented, and verified through simulation, four synthesis algorithms, including: hierarchical actor structure, replication, finite state machine merging, and a global coordinator. The algorithms were implemented in the ROOM synthesis engine of the MESA CASE tool [5, 6], an ongoing development project in our software requirements engineering research group. I drew, verified, and synthesized all MSC's using this tool. I also performed various development and maintenance activities on it throughout the work.

The synthesis techniques presented have been successfully verified to work on a reasonably complex telephony system which I partially built.

I have also investigated the synthesis of Promela models from MSC's and shown how to validate safety and liveness properties of the original specification within the constraints of limited available memory. Improvements to existing algorithms have been suggested and implemented, using a GSM system that I built from scratch as a testbed, and of which I identified a number of properties to prove. The system was then synthesized into a ROOM model using the ideas of the previous chapter.

Chapter 2

Background

Models are a way of capturing the knowledge of a system. They describe its elements and their interrelationships. Control and data flow is clearly specified. An important characteristic of models is precision, attained through rigorous mathematical semantics. Models can present multiple views of the system, including its physical structure and behavior, and define its causality of events, concurrency, and synchronization [18]. Several modeling languages are prominent in the software engineering field, and are summarized below.

2.1 MSC's

MSC's (Message Sequence Charts) are a popular and easily understandable visual specification language for system engineering. They consist of a graphical and equivalent textual language for the specification of functional requirements for concurrent, reactive and real-time systems. MSC's are standardized by the International Telecommunication Union in Recommendation Z.120 [55, 56]. They were specially invented to describe signal exchanges between system components executing in parallel [8], and are to a large extent design-independent in that they can be implemented through

many software architectures. MSC's can also be used to describe software architectures; for example, through patterns [1].

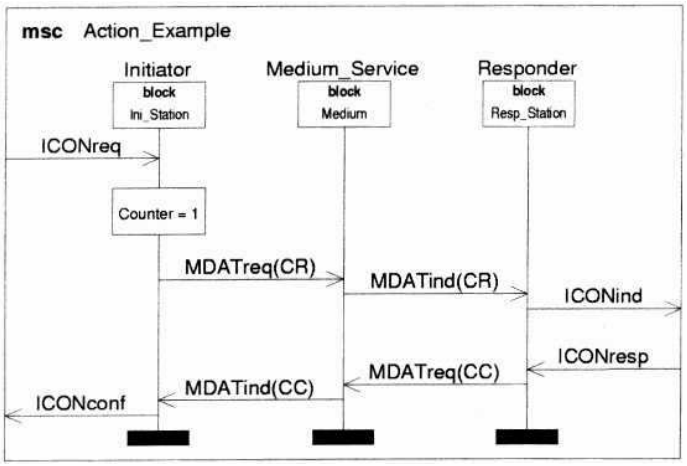

Figure 2.1: An example of a basic MSC, defining signal transmissions between three process instances and the environment. The counter initialization block is only treated as a comment and has no syntactic meaning [8].

The MSC-92 standard [35], approved at the ITU meeting in Geneva in 1992, introduced instances, messages, events, and conditions to simple sequence charts. The synthesis work herein is primarily based on the MSC-96 [36] version of the specification, approved at the conclusion of the last study period in April 1996, which added features such as hMSC's, references,

[1] A pattern is a recurring solution to a design problem [38]. It consists of a set of rules describing how to accomplish certain tasks. The rules express a relation between a context, a system of opposing forces that occur repeatedly in that context, and a software configuration where these forces are resolved. The forces must balance, or trade-off. Patterns reflect the knowledge of and experience in software development. While patterns are the instructions for implementing solutions in software, frameworks are the physical realization of multiple software pattern solutions. Architectural patterns, in particular, address software architecture. Architectural patterns express a structural organization schema, provide a set of predefined subsystems, specifying their responsibilities, and include rules and guidelines for organizing the relationships between them. MSC's can be used to specify the communication between subsystems as part of a pattern.

general ordering, and in-line expressions. The very recent MSC-2000 [37] recommendation, approved by the ITU-T Study Group in November 1999, has introduced improved structural concepts and object-orientation, data, time observations and constraints, and synchronizing method calls [66, 37].

MSC's have a number of uses:

- The documentation of system requirements or message exchange among concurrent objects.
- The description of test scenarios.
- The expression of properties verified against SDL specifications.
- The capture of observed behavior of a working system or a simulation of the system specification or design-time model.

The first area is of primary interest in this work, as it constitutes the input to the synthesis and design work of the system. Figure 1.1 on page 3 illustrates how MSC's are integrated into the waterfall development process [5]. They are created based on requirements elicited from the customer, and analyzed for correctness. They imply the process topology and communication protocols required for drafting an architecture of the system. MSC's can also be used to generate test cases to validate correct operational behavior of the final implementation by comparing execution traces to the original specification. Message sequence charts are useful for the documentation of functional and interface requirements. They have found use in the design of both software and hardware systems.

MSC's consist of a set of individual system traces called scenarios that describe partial ordering of input and output events: message sending and receiving actions. The scenarios rely on existential quantification[2] [22]. MSC's are inappropriate for the specification of data storage and manipulation, as

[2] i. e. The scenarios depict events that can occur in the system, rather than all of the events that must occur.

well as processing logic. Although extensions exist, MSC's are specifically intended to address message events.

Although this is not part of the Z.120 standard, MSC's can be used to describe exceptional and mandatory occurrences, those that must or must not occur in all runs. Such extensions are supported in annotated MSC languages such as Live Sequence Charts, described in 3.3.

The fundamental chart in the language is a *basic* (existential) Message Sequence Chart (bMSC), which essentially depicts a partial description of behavior of a set of process instances. An example appears in Figure 2.1. The process instances run in parallel and exchange messages in a one-to-one, asynchronous manner. Processes are represented by vertical lines, such as the `Initiatior` in the example, while the ends of the arrows denote message send and receive events. SDL systems, blocks, processes, or services can map directly to MSC process instances. The environment is an entity that is represented by the enclosing rectangular frame on the MSC diagram. It is treated much like a process instance, and can also send or receive messages, with the exception that no ordering of communication events is assumed. A system which interacts with the environment is characterized as an *open system*.

MSC's can specify just one sample system execution, illustrating a possible run. They can also specify all valid and invalid behaviors. Applying the *closed-world assumption* to MSC's, it would be theorized that any MSC not included in the system description as an allowed or disallowed behavior cannot occur. However, this assumption does not hold true of MSC specifications, as unspecified traces may still be possible.

Processes communicate through message exchanges, indicated by connecting arrows along a vertical time axis, such as the `ICONreq` message sent by the environment. Time is represented down the length of each process axis, and so a total ordering of communication events is shown from top to bottom in a single chart. In addition to message exchanges, processes can individually execute internal actions, use timers to express timing con-

straints, and create and terminate process instances. Timers can be used to specify timing constraints for hard real-time systems. Co-regions can also be specified, where the ordering of the actions within the boundaries of the region is undefined. Conditions, defined for one, some, or all processes (local, non-global, or global conditions) are special marker states that can be optionally specified for the purpose of composition (combination) and decomposition of MSC's.

MSC's do not inherently assume a communications channel and buffer for transporting and storing messages, hence no properties of the communication medium can be asserted. The MESA tool does allow for the definition of channels, but not in compliance with the Z.120 standard. An underlying error-recovery protocol, such as one that uses timeouts and acknowledgments, is normally assumed to be present for most networking protocols but is left unspecified, so that messages are assumed to always arrive correctly. A construct for a lost message can be used if required, though.

It is possible to refine a process instance by a set of instances defined in another MSC, which is then called a *subMSC*. Messages addressed to the refined instance appear internally as messages connected to the subMSC's border, with the order preserved. This concept makes it easier to manage highly populated specifications by grouping process instances together.

Although the textual and graphical formats of MSC's contain the same information, the representation of MSC's is different in the following respects [20]:

Textual representation of MSC's is *instance-oriented*. All events are defined as being ordered per instance. Each message input is not directly identified with an output.

Graphical representation is *event-oriented*. Message inputs and outputs are always depicted as being connected, so that it is easier to visually interpret the protocols.

A number of basic MSC's can be connected in a directed graph in order

to describe parallel, sequential, iterating, and non-deterministic execution. The resulting diagram is called a *high-level* message sequence chart (hMSC), as shown in figure 2.2. An hMSC is a digraph, where all nodes can refer to bMSC's, and the edges indicate execution sequences possible along the nodes. The nodes can also describe a system in a hierarchical fashion by combining multiple, connected hMSC's within a single hMSC. An hMSC has exactly one *start* node ($\bigtriangledown$) to denote the beginning of the specification, and optional *end* nodes ($\bigtriangleup$) to denote the termination of the specification. A directed line between two nodes of an hMSC indicates that they are composed vertically, and more than one outgoing line from a node implies a non-deterministic branching choice, where the successors are alternatives. A cycle connecting a number of nodes implies a possible repetition, or loop.

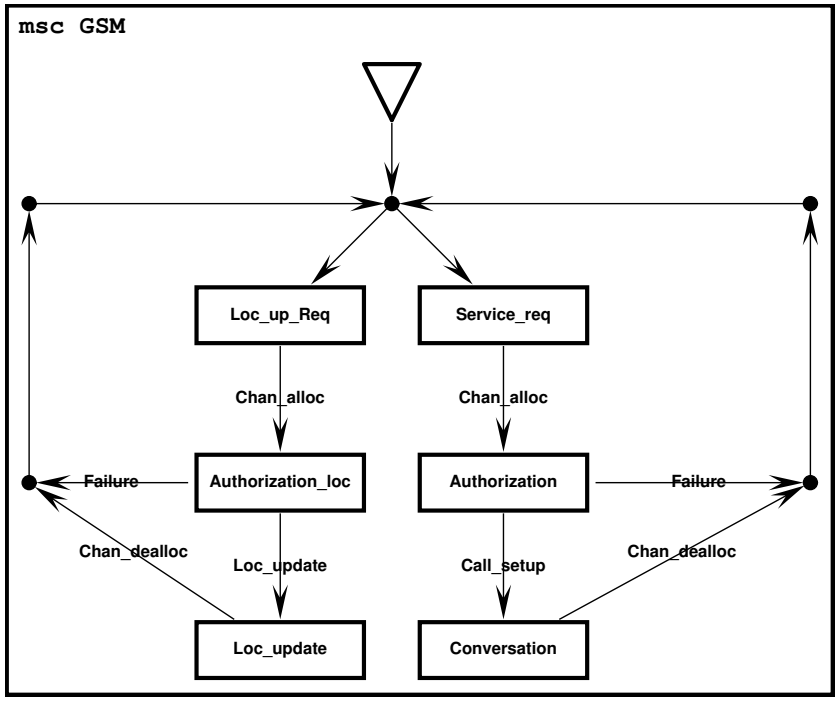

Figure 2.2: An hMSC of the GSM Promela Model from Chapter 5.

MESA, the Message Sequence Chart Editor Simulator and Analyzer, is a

CASE tool that provides an environment for the composition of system models using the MSC syntax and semantics, supporting syntactic and model-based analysis. MESA, developed in the Dept. of Electrical & Computer Engineering at the University of Waterloo, was used for the entry of the PBX and GSM models described in the text. Non-local branchings and process divergence were identified by the tool, and its synthesis features was used to translate the MSC's to ROOM and Promela code.

2.2 ROOMcharts

ROOMcharts (Real-Time Object-Oriented Modeling) methodology was developed as an aid in the design of distributed real-time systems, and is based on an object-orientation paradigm [3]. ROOM actually uses a variation of the visual formalism of basic statecharts (discussed in Section 3.2) called ROOMcharts. It is implementation-focused, in the sense that careful attention has been given to the definition of features that can are directly realizable in a software engineering design tool.

ROOMcharts consist of hierarchical communicating extended finite state machines. They ensure consistency and clarity in the design model by specifying a formal conceptual base. ROOMcharts permit efficient implementations while still retaining significant expressive power [11]. ROOMcharts expand on statecharts by formally defining structural boundaries and interfaces. In addition, ROOMcharts make use of a message-passing paradigm, which can be implemented in a distributed system, rather than the more vaguely defined and less practical broadcast system suggested in Statecharts, in which a change of state can be conveyed to any component instantaneously.

With respect to orthogonal (concurrent) states, ROOMcharts do not incorporate them, as they can usually be replaced by concurrent and connected communicating actors. This avoids extra complexity and potential undesirable couplings between states.

ROOM models are multi-faceted. They are composed of the following distinct views of a system:

Structural The physical layout of the system is expressed in terms of the instantiation of concurrent actors (active system components, usually software processes) and the statically defined interfaces between them. The interfaces consist of ports, through which all messages must be exchanged. A protocol is an attribute of a port, and consists of a set of valid message types that can pass through a port. Bindings specify the connectivity of interfaces between actors. Each process possesses exactly one input queue for all incoming messages destined for it. Classes of actors and protocols may be defined, inheritance relationships may be specified, and a hierarchical topography may be introduced into the structure, as well. The decomposition structure of the system can be regarded as being an architectural view of it. An example of the structure of a ROOMchart is shown in Figure 2.3.

Behavioral The description of the actions undertaken by each actor in the system is described in a state-based behavioral view. Two approaches to message communication are supported: synchronous (in which the sender blocks waiting for a reply to its message before proceeding) and asynchronous (in which the sender immediately continues processing after sending a message without waiting for a reply). An actor is event-driven, in that it normally sleeps waiting for input then immediately responds to it. Each actor possesses a single thread of execution, and run-to-completion semantics[3] are utilized. If a new event occurs

[3]An event is said to occur when a message is received by an actor. In the *run-to-completion* approach, the processing performed by an actor that occurs due to an event cannot be interrupted by an event of higher priority, i. e. while an actor is busy handling a message, the arrival of a message of higher priority than that of the current one cannot interrupt the actor — it will be queued up and handled later, on completion of the current event. This is opposite from the *preemptive* model, in which the arrival of a higher-priority message causes the processing of the current event to be suspended and commenced with the new event. The complication with the latter method is that it leads to a concurrency problem with respect to internal variables. In the run-to-completion approach, a higher-

while an actor is still busy processing the previous one, the new event is queued by the receiving end port and will be resubmitted once all prior messages have been processed. Messages can be prioritized, but this is treated as an implementation issue. The behavior within actors is specified by hFSM's (Hierarchical Finite State Machines), with a complete description of all states, including the actions taken on transitions, in the form of executable code. A memory is maintained for states so that execution returns to the most recent state when an actor is re-entered during execution. Trigger and guard definitions determine how each transition is enabled. Provisions for inheritance of behaviour by refining the leaf states of a parent class are also available. An example of the behavior of a ROOMchart is shown in figure 2.4.

It is unwieldy to represent all of the transitions in the entire system in one monolithic actor, and so, through the process of modularization and encapsulation, all of the tasks are usually divided into coordinated actor components, with the internal finite state machines of actors also assuming a hierarchical arrangement.

The communications model makes the following assumptions that are common to channels in Communicating Finite State Machines (CFSM's)[4]: full-duplex, error-free, first-in first-out (FIFO), and unbounded capacity.

priority event can, however, claim the processor and cause a context-switch if the recipient actor is waiting for a message to arrive when the new event occurs rather than processing one.

[4]CFSM's [23] introduce the concept of communicating channels to finite state machines (FSM's). Each FSM represents a communicating, concurrent process, with a finite number of control states.

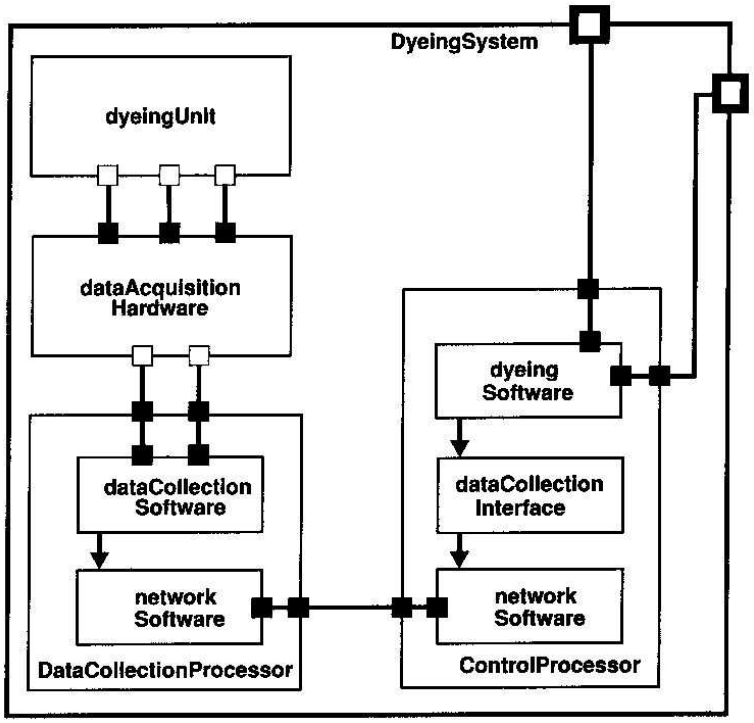

Figure 2.3: An example of the structure of a ROOMchart [3] for a fabric dyeing system, containing the class definition for DyeingSoftware. The square shapes on the rectangular actors denote end ports (known precisely as port service access points, or port SAP's), as they are attached to the ultimate producers or consumers of the messages. The hollow squares on the boundary are relay ports, allowing the encapsulated components to export their interfaces. The relay ports simply act as intermediaries to pass messages. The arrows denote communication through an alternate mechanism called system service access points (system SAP's), supporting message-sending across vertical structural layers, with the arrowhead pointing towards the lower layer.

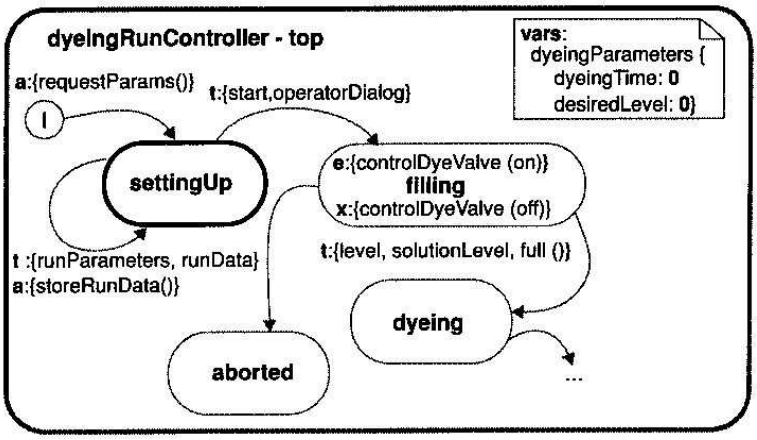

Figure 2.4: An example of the behavior of a ROOMchart [3] for a fabric dyeing system, with its finite state machine shown. The annotations have the following meanings:

t: enabling transition

a: action taken during transition

e: action taken during entry into state

x: action taken during exit from state

The arc from the circled I is the initial transition.

Chapter 3

Related work

The following sections present related work on specification languages and synthesis.

3.1 SDL

SDL (Specification and Description Language) [25] is a high-level formal programming language represented in graphical (SDL-GR) or textual syntax (SDL-PR), standardized by the ITU-T in Recommendation Z.100 [19]. It is an industry-caliber language used in the development of network and telecommunications systems, where it is considered the de facto standard.

SDL is presented here as it is often used in close combination with MSC's, where MSC's can be used to express and verify properties of SDL specifications, and for other purposes described below. SDL complements MSC's in producing a full system description.

SDL describes a system as a collection of concurrent communicating processes, each modeled as an extended, hierarchical finite state machine (EFSM). It is richer than MSC's in that it supports structural concepts such as processes contained within blocks and connected by channels and

signal routes. It supports abstract data types, inheritance, instantiation, and parameter passing. The content of tasks and decisions may consist of knowledge within the scope of the problem domain, but not necessarily defined within the context of the SDL specification itself, and as a result is not parsed. MSC's have no concept of data flow or processing activities, other than a process's behavior being dictated by what message is next expected in a specific chart. Both models do support the notion of non-determinism.

The communications model is well-defined, consisting of each process responding to a bounded first-in first-out input message queue. A maximum number of instances of each process type must be statically defined, although dynamic allocation and deallocation is allowed. MSC's, however, are more generic, with no such physical limits being defined, at least in MSC-96.

Unlike SDL, MSC's in general are not entirely adequate for use as a complete system description. Indeed, part of the motivation behind the creation of MSC's was for their anticipated adoption as a language complementary to SDL, in fact standardized by the same ITU-T study group. MSC's may be used in the automatic generation of skeleton SDL specifications, as well as for the simulation and consistency verification of SDL.

An example of a structural and a behavioral SDL diagram appears in Figures 3.1 and 3.2.

3.2 Statecharts

Statecharts are a generalization of state transition diagrams [10, 15, 18]. They integrate three concepts: transition diagrams, depth, and orthogonality. Depth involves the clustering of states into higher entities called superstates, a process referred to as multilevel decomposition, and hiding the illustration of internal details, through a technique called *zoom-out*. By incorporating such graphical constructs for finite state machines, the apparent complexity of the system is reduced. An example appears in figure

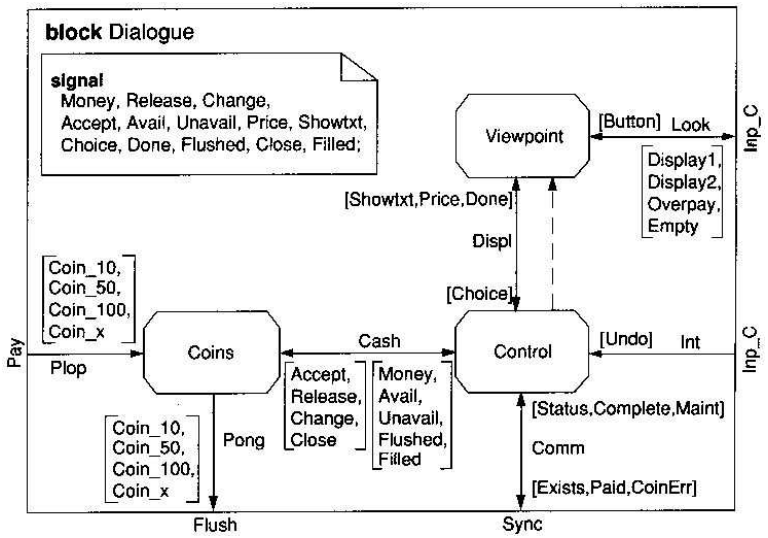

Figure 3.1: SDL structural example of a candy vending machine [25]. Instances of process types communicate through signal routes and channels, and are laid out in hierarchical fashion in layered blocks. The environment is denoted by the enclosing frame.

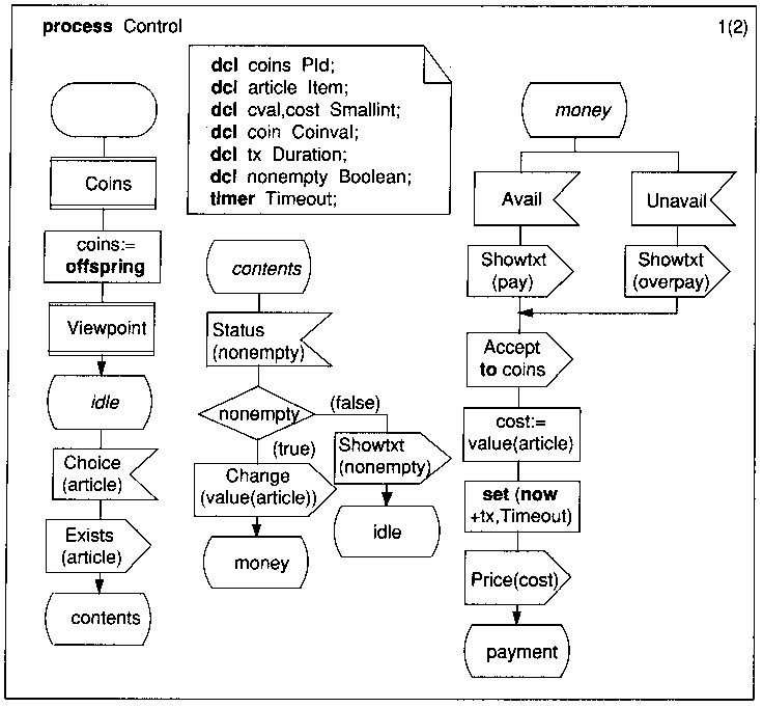

Figure 3.2: SDL behavioral example of a candy vending machine [25]. The various constructs denote states, signal transmissions and receptions, procedure calls, decision points, timer activity, and tasks. Every variable used must be declared.

3.3, illustrating the state machine of a statechart and the semantics of its transitions.

As explained in Section 3.3, orthogonality is the concept of a process instance being in two states at one time, each state defined in an orthogonal component. It is possible for a statechart to be split into multiple state machines, with the encapsulating state representing the orthogonal product of the components. That is, each component consists of independent sub-states, and it is possible for a single event to trigger simultaneous transitions within the state's components.

Module-charts are used as a form of data flow diagram. They describe modules that constitute the implementation of the system, including the decomposition of the system into software and hardware components, and the messages exchanged between them.

A *history* is maintained for each component so that the most recently visited state is immediately entered on a new transition into the component. Thus, the behavior of a system is not memoryless. MSC's, however, do not support transitions across levels inside of charts, so that history is not a concept that is applicable to MSC's. As a consequence, the intermediate exiting of a chart, in which a higher-level transition triggers an immediate exit from all sub-components of a state, is also not supported. All events in a bMSC must always run to completion, until the end of the bMSC is reached, and no interrupts are possible. Also, the system may only be in one bMSC at a time, unlike the orthogonal components of statecharts.

The communications model consists of a form of event broadcasting. Internally generated events are immediately reacted to by all orthogonal components of a state. However, the mechanism is limited to the scope of a single statechart, and thus takes no part in inter-object communication. MSC's do not support the notion of a broadcast at all. Each message must have a single, explicit recipient.

Non-determinism occurs when several transitions that cannot be taken simultaneously are enabled, and no added criterion has been given for se-

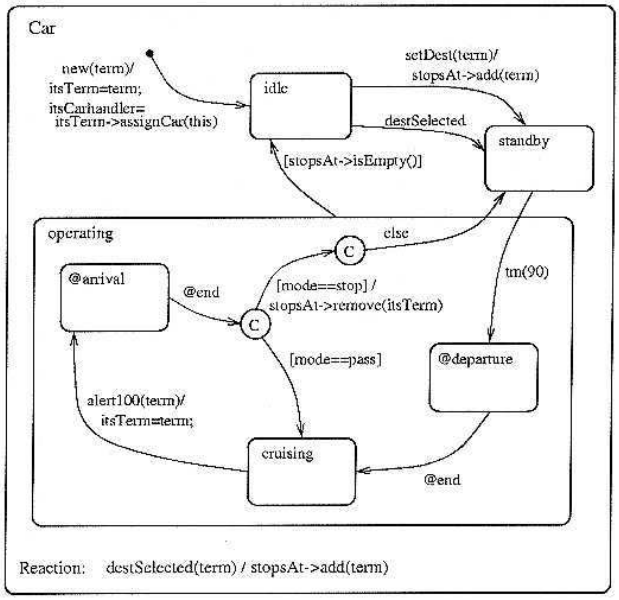

Figure 3.3: A statechart of a rail-car system [15]. The transition arrows are adorned with reactions of the form: `trigger[condition]/action`. Events always run-to-completion, as in ROOMcharts (see Section 2.2 on page 30). The initial transition appears in the top left-hand corner (•). The `@` symbol prefixing a state name indicates that it is decomposed into another sub-chart. Consider the transition labeled `[stopsAt->isEmpty()]`: if the car stops at all stations, it will be removed from the `stopsAt` list (through the `remove` command specified inside of the `operating` sub-chart). Control will exit from the `operating` sub-chart and into the `idle` state, regardless of which state the `operating` sub-chart was in when the condition was satisfied. The sub-chart does not constitute an orthogonal component, which is indicated, instead, by a sub-chart being divided by a dotted line into parallel state machines. No history is defined for the charts shown, as evidenced by the lack of a circled `H` symbol. If it were present, and the transition labeled `tm(90)` pointed to this symbol, then whenever the `operating` sub-chart would be re-entered from the higher-level `Car` chart, control would return to the most recently visited state. In this case, such a change would be illogical. ©IEEE Press, 1996.

lecting one of the available choices. Tools that execute the model must make an arbitrary decision or ask the user to decide. Unlike in MSC's, however, prioritization can cause certain (but not all) situations to be deterministic. For instance, a higher-level transition has priority over an internal transition within a substate. MSC's do not support the concept of signals being prioritized.

Transitions are more general than those in MSC's — they can depend on whether the system is in another specified state, and can be triggered by the entry or exit to and from another state.

The STATEMATE toolset provides a graphical editor for the entry of statechart models. It checks for syntax, consistency, and completeness. It allows external code to be attached to the model. Models can be executed and animated, and liveness properties are validated through exhaustive execution of models. The models can be translated to either Ada or C code, and a variant is available for translation to VHDL/Verilog. STATEMATE supports rapid throw-away prototyping[1].

Executable object modeling is supported under C++, where the communication between statecharts assumes an event-queued client-server environment [16]. The support tool is called Rhapsody Modeler^{TM}and Developer^{TM}, available from I-Logix. A single-threaded approach is taken, so that only a single reaction to an event can occur at a time, i.e. a transition cannot take effect until all orthogonal components are waiting in states. Full broadcast is supported, accomplished by addressing an event to the system object. Both synchronous and asynchronous communication is supported. Classes are specified in the object model, and the notion of instantiation of objects is included with additional notation denoting the number of instances. Instantiation of typed statechart behavior is supported with limited refinement

[1]Throw-away prototyping is a methodology whereby code is developed to explore factors critical to the system's success, such as the ability to meet performance constraints, and then that code, usually not produced in a maintainable state, is discarded after analysis. The use of a rapid prototyping tool allows for much faster creation of a prototype than would be the case with a standard manual implementation [7]

possible, including decomposition into substates or orthogonal components.

3.3 LSC's

The synthesis of object systems from LSC (Live Sequence Chart) specifications has been investigated by Harel et al [9, 14, 46]. LSC's are an extension of MSC's that address their limitation in expressive power. For example, although the order of message sending and receiving can be specified in MSC's, they cannot express what scenarios (called *anti-scenarios*) are forbidden in the system.

The principal additions are liveness properties, which specify mandatory (called "hot"), eventual behaviour in the system, as well as forbidden scenarios.[2]

Parts of a chart or even the whole chart itself can be specified as being mandatory or provisional. Live Sequence Charts (LSC's) can express whether a communication, instance progress, or a whole chart will be involved in all runs or in some runs. The two types of charts that can be specified are as follows:

Universal Whenever an activation condition is satisfied, the system must follow the behavior specified in the chart, i. e. the sequence of messages in the chart must occur in the specified order. An activation condition can take the form of a single message or a sequence of messages defined in a chart. Therefore, a causal precedence relationship exists between the activation condition and the chart.

Existential The charts need not be satisfied in all runs. It is only required that for each of these charts, there is at least one run in the system

[2]Liveness properties specify an event that must or must not happen when a certain condition is true. For instance, an example of such a property is that if a phone goes off-hook, then a dial tone is emitted. The system must satisfy this property in order to be judged as being correct with respect to the requirements.

that satisfies it. Existential charts can always be transformed into universal charts specifying the exact activation message or prechart that is to determine when each of the possible approaches occurs.

During execution, if a false mandatory condition is encountered in a system run, the run aborts abnormally, while a false provisional condition implies a normal exit from an enclosing chart or subchart.

LSC's are essentially annotated MSC specifications, and are as expressive as statecharts. The underlying automaton is called a *life Büchi* automaton [9]. A standard non-annotated MSC specification, on the other hand, corresponds to a safety Büchi automaton, in which only pure safety properties can be validated [3].

Message sequence charts, on the other hand, describe only the order in which messages may be sent and received. They can only convey safety properties, those that must be currently satisfied for a given state, not at any point in the future, and these properties are limited to only the order of message send and receive events.[4]

The validation of liveness properties involves a *never automaton*, derived

[3]The Büchi automaton consists of a set of states, an initial state, an event alphabet, a transition relation for specifying state changes due to input events, and a set of acceptance states. The automaton accepts an infinite sequence of events if it causes the automaton to visit at least one of its acceptance states infinitely often. The Büchi automaton is more general than a finite automaton, which accepts a word if it causes the automaton to halt in one of its acceptance states after a finite trajectory on the word. A Büchi automaton for which every state is an acceptance state specifies a pure safety property, and no liveness. Liveness is only expressed if a non-empty subset of the states are not acceptance states.

[4]An example of a safety property is that it is impossible for an idle tone to be emitted at the same time as a dial tone (represented by the appropriate signals). This follows from all of the possible scenarios encoded in the MSC. A more general safety property is that no deadlocks are present in the system. However, a property such as a phone not receiving a dial tone when being taken off-hook due to all touchtone receivers (TTRX's) being in use cannot be expressed in MSC's. Nor can the property that once a connection is established, either the originator or receiver will eventually terminate the call.

from the Büchi automaton[5]. It must be ensured that an acceptance state corresponding to the fulfillment of an illegal property (the opposite of a safety or liveness property) is not part of a sequence of states that can be repeated infinitely often [17].

Activation conditions in LSC's may be defined that specify the scenarios to which the system's behaviour must conform when the specified conditions are satisfied. Through a combination of different message line orientations and boundary boxes, mandatory and provisional scenarios are visually depicted. An example is illustrated in Figure 3.4.

In addition, Harel's state machines are orthogonality-free and flat, while provisions are made for the synthesis of more advanced constructs. *Orthogonality* is the characteristic of a process being active in more than one state at a time. In other words, it entails concurrency. Essentially, Harel et al's paper critiques that the current synthesis employed in the MESA tool is based on the limiting assumption that all bMSC's are mutually exclusive of each other such that the system can only be in one state in one bMSC at one time. Different bMSC's are executed at different points in time[6]. While the concept of orthogonality cannot be expressed in MSC's, this thesis presents a solution to behaviour specified in a hierarchical arrangement, not simply a flat state space, an idea that is not expressed in the work on LSC synthesis.

LSC's provide constructs for specifying the entire chart, events, conditions, and instance progress as universal or existential for a set of runs. However, they don't provide explicit representation of precedence and causality relationships between the events in any execution. For example, these charts can express that there exists a run `r1` where event `e1` will happen, and there

[5]The property to be proven to hold true of the model is first inverted, then translated into an automaton. Next, the synchronous product is built of the model and this automaton. The verification consists of showing that there is no trace that ends in an acceptance state.

[6]This treatment is, however, consistent with usage in industrial MSC tools, such as Lucent's μBet (Lucent Behavior Engineering Toolset). μBet represents behavior as hierarchical use cases containing scenarios, with the underlying scenario behavior being represented by MSC's.

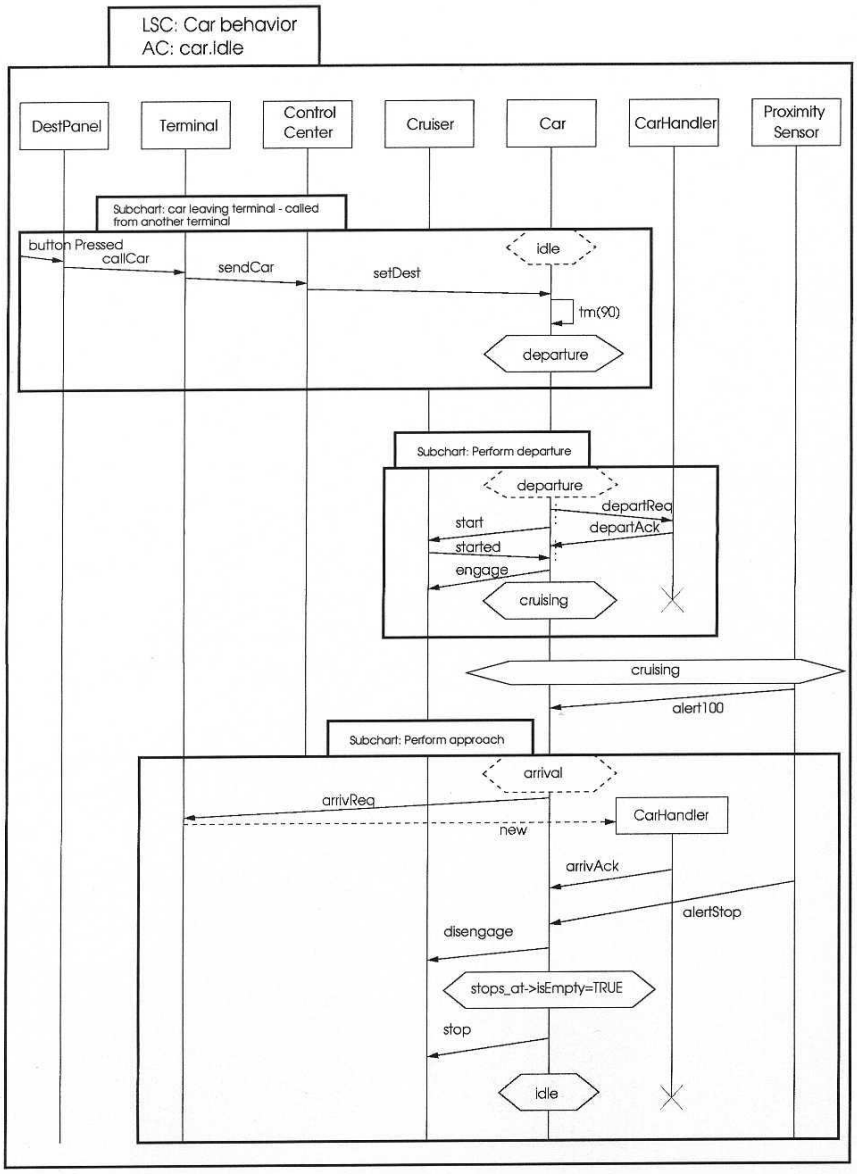

Figure 3.4: Live sequence chart example of a rail car. The sub-charts confine the instances relevant to them, while the others are transparent. The sub-charts have solid borders and are thus universal, while existential subcharts (requiring only one satisfying run) are indicated by dashed borderlines [14].

also exists a run `r2` where event `e2` will happen [7]. However, they cannot express the relationship between the occurrences of the above two events. More precisely, these charts cannot express: if in a specific run `e1` happens, then the occurrence of `e2` is mandatory in that same run.

The expressivity and verification of these action/reaction types of properties in a particular run is of interest here, however, and temporal logic (see 5.5 on page 154) allows for the expression of these properties. In this work, LTL has been used to specify properties on the Promela model generated from an MSC specification that the MSC specification cannot itself express. A Promela model can be simulated and validated for safety and liveness properties using the Spin tool [51].

LSC's would be insufficient to express these specific types of liveness properties. For instance, consider the requirement in Subsection 5.5.1 on page 158, where after being identified and having its subscriber information retrieved, the mobile must be authenticated by the network, with the result being success or failure. This rule implies that the identification event must be followed by an authentication event in the same run, and both events must occur, but this cannot be expressed by LSC's, as explained above.

3.3.1 LSC object synthesis

The synthesis of LSC's addresses the problem of whether, given an LSC specification, there exists a satisfying object system and, if so, synthesizing it automatically [9]. An entire LSC specification is considered *consistent* if and only if it is satisfiable by a state-based object system. A satisfying system is then synthesized as a collection of finite state machines or statecharts. Live sequence charts are considered a manifestation of use cases, and if they can be synthesized, can directly lead to an implementation.

A crucial difference between the LSC synthesis and this work is that the former allows for the specification and synthesis of liveness properties.

[7] `r1` and `r2` may be the same run.

The MSC language does not support this concept, and hence only possible paths (scenarios) are synthesized with no liveness dependence among them. However, *progress labels*[8] and *acceptance conditions*[9] can be used with the Büchi automaton, which forms the basis of MSC specifications.

The LSC synthesis uses a more restricted form of the LSC language as its basis, in which objects only communicate synchronously, i. e. the sender of a message is blocked waiting for a receipt. This thesis, however, addresses asynchronously communicating state machines[10], which can be more efficient, as explained in Section 1.4 on page 8.

An additional restriction is that communication with the environment can appear only as an activation condition of an LSC chart and not as part of the messages in the chart itself. There is no such restriction on the MSC's that are synthesized as described in this thesis. The environment is essentially treated as a unique process instance.

Much of the LSC work deals with state machines which are orthogonality-free and flat, while the thesis examines systems with hierarchical structure and behavior. However, a more complex algorithm is outlined for synthesizing statecharts with concurrent, orthogonal state components.

The ROOM synthesis from MSC's produces a result similar to that of statecharts, in which an actor is synthesized for every MSC process instance, and all actors run concurrently. However, there is no concurrency within each actor, i. e. an actor cannot consist of orthogonal components as it can only occupy one state at a time. While the algorithms for LSC object synthesis also do not support the notion of orthogonality, an addition is

[8] An invalid cyclic execution sequence is defined to be a sequence of statements that can be repeated infinitely often, without achieving any progress in the execution of the system. It it possible to specify precisely which states constitute progress by attaching labels to states [17].

[9] Acceptance states are the opposite of a progress condition — they formalize that something *cannot* happen infinitely often. They mark the states that may not be part of a sequence of states that can be repeated infinitely often [17].

[10] A synchronous communication exchange can always be represented in an asynchronous model, if needed.

roughly sketched out such that each object in the system has a top-level state and any number of orthogonal components. A general description of this addition to the synthesis algorithms appears at the end of this subsection.

One of the most difficult aspects of LSC synthesis is the interaction between statecharts, in which the behavior of a synthesized statechart is dependent on the requirements of other statecharts. The synthesis of MSC's does not deal with interdependent MSC's. The entire system is always in a single state governed by the behavior specified in a single bMSC. This chart specifies the behavior of all process instances in the system during its lifetime. Thus, MSC semantics assume mutually exclusive charts. In the ROOM model synthesis, the information used by the synthesis algorithm pertains only to those messages directly relevant to each process in question, i. e. in which the process is either a sender or receiver. However, because the LSC specification language allows both existential and universal quantification and does not assume mutual exclusion semantics, this projection information is insufficient.

Just as with MSC's, the dynamic creation and destruction of LSC instances is not considered in this thesis. Also, a common assumption is that no failures are present in the system, and hence every message that is sent is immediately received.

In LSC synthesis, the notion of a *cut* is defined, representing the progress each instance has made in a scenario in terms of event locations (situated on message send and receive actions). The sequence of cuts in the execution order constitutes a *run*, essentially denoting the sender of each message and the identifier of the message itself in a possible execution order. An example of a cut and a trace appears in figure 3.5 on page 52. A chart may have multiple runs, characterized by multiple message event sequences, and they may be executed in any order. In MSC's, all process's can only occupy a state defined by each process's send and receive events only in the same bMSC. In the Ladkin/Leue semantics for MSC's, a synchronous communication event of sending and reception likewise corresponds to exactly one transition in the global state transition graph [39].

In the example of figure 3.5, the `setDest` message is the activation condition for the chart. An example of a single cut is:

$$(< cruiser, 1 >, < car, 3 >, < carHandler, 2 >)$$

where execution has reached location 1 in instance `cruiser`, location 3 in instance `car`, and location 2 in instance `carHandler`. This corresponds to the `start` message being sent from the `car` to the `cruiser`. At each location, a send or receive event occurs. As the instances communicate with each other, only certain combinations of positions, i. e. cuts, are possible. An example of a single possible run is:

$$(env, car.setDest)$$

$$(car, carHandler.departReq)$$

$$(carHandler, car.departAck)$$

$$(car, cruiser.start)$$

$$(cruiser, car.started)$$

$$(car, cruiser.engage)$$

The automaton for this chart would be represented by a linear series of nodes, beginning with node n_0, representing cut (0,0,0). A transition from n_0 to n_1 is labeled with the message sent in the event `departReq`. n_1 represents cut (0,1,1), and makes a transition to n_2 to cover the `departAck` message, and so on. If orthogonal components are present, then the automaton would contain branches so that every possible ordering of events in the chart was depicted. An example of this is included in [9].

For a universal chart m, the DFA (deterministic finite automaton) accepting its language is defined as:

The transition relation is formally defined as follows [9], with additional comments:

$\mathcal{A} = (A, S, s_0, \rho, F)$ where:

- $A = A_{in} \cup A_{out}$ is the alphabet.
- The set of states S consists of the *cuts* through m, with an additional state s_0, so that: $S = \{c | c \in cuts(m)\} \cup s_0$.
- Assuming the natural mapping f between $(dom(m) \cup env) \times \Sigma \times dom(m)$ to the alphabet A, the transition function ρ is defined as follows:
  - $\rho(s_0, a) = c_0$ if $a = f(amsg(m))$ and c_0 is the initial cut, i. e. Make a transition to the first cut if the first event is the chart's activation message.
  - $\rho(c, a) = c$ if a is not restricted by m, i. e. Return to the same state (as a self-transition) if the event is not mandatory.
  - $\rho(c, a) = c'$ if $succ_m(c, < j, l_j >, c'')$ and $succ_m(c'', < j', l'_j >, c')$ and $f(msg(m)(< j, l_j >)) = a$ and $< j, l_j >, < j', l'_j >$ are send and receive events of the same message and not all locations in c' are cold, i. e. Make a transition to the next cut on a single send-and-receive pair event.
  - $\rho(c, a) = s_0$ if $succ_m(c, < j, l_j >, c')$ and $f(msg(m)(< j, l_j >)) = a$ and all locations in c' are cold, i. e. return to the initial state once all messages restricted by the chart are received.
- The set of accepting states is $F = \{s_0\}$.

The underlying automaton representation for all possible runs in LSC's, which is used for the construction of the synthesized state machines in the synthesis algorithms, consists of nodes of each state representing a cut and being labeled by the vector of locations, with edges between the nodes labeled with the message sent. Only send events are represented, since a synchronous communication model is assumed (again, unlike in the synthesis of MSC's). The LSC language enables the forcing of progress along an instance line as part of the "liveness" extensions. Each location is assigned a "temperature:" *hot* or *cold*. Any run must continue down hot lines, while

it may or may not continue down cold lines. In the final cut of a run, all locations must be cold.

The target object system is a basic computational model for object-oriented designs, defining the behavior of systems composed of object class instances whose behavior is given by conventional state machines [24]. A system *satisfies* an LSC specification if, for every universal chart and every run, whenever an activation message holds, the run must satisfy the chart, and if, for every existential chart, there is at least one run in which the activation message holds and the chart is satisfied. To show satisfiability, a *GSA* (Global System Automaton) is constructed, describing the behavior of the entire system in terms of the message communication between the objects in the system in response to messages received from the environment. The GSA is a finite state automaton with an input alphabet consisting of messages sent from the environment to the system, and an output alphabet consisting of messages communicated between the objects in the system. The construction of a GSA satisfying the specification implies the existence of an object system (where a separate automaton is created for each object) satisfying the specification. It is possible that a given LSC specification is not satisfiable by an object system. This can occur due to the interaction between more than two universal charts, and also when a scenario described in an existential chart can never occur because of the restrictions from the universal charts.

A universal chart must be satisfied by all runs from all points in time. Consistency of an LSC specification is a necessary and sufficient condition for the existence of an object system that satisfies it. An algorithm is provided to test for consistency. The GSA is then used as part of the synthesis by distributing it between the objects, creating a desired object system. Three approaches are possible:

Controller. A single controller sends commands to all objects so that each object acts in the desired fashion. The size of the state machine of the controller object is equal to that of the GSA.

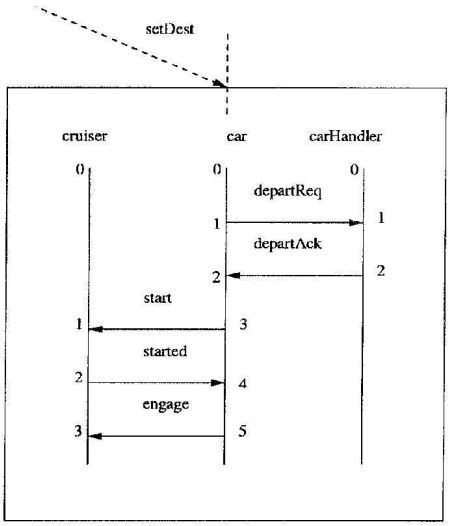

Figure 3.5: A single LSC of an automated railcar controller system is shown.

Full duplication. There is no controller object. Each object has the state structure of the GSA, and thus "knows" what state the GSA is in. Each object is therefore autonomous.

Partial duplication. The GSA is distributed as in the full duplication construction, but states that carry information that is not relevant to the object in question are merged. The automaton is similar to that generated for MSC synthesis.

An automaton is thus created for each LSC, the transitions representing the message events, and states for activation conditions being labeled as acceptance states. An intersection automaton is then created depending on the approach chosen.

Unlike the synthesis of ROOM models from MSC specifications, the result of the synthesis of LSC's is an unrefined generic object-based model.

The three approaches described above are illustrated and explained in figures 3.6-3.8, re-drawn from the examples in [9]. A comparison to the MSC synthesis approach is included in figure 3.9.

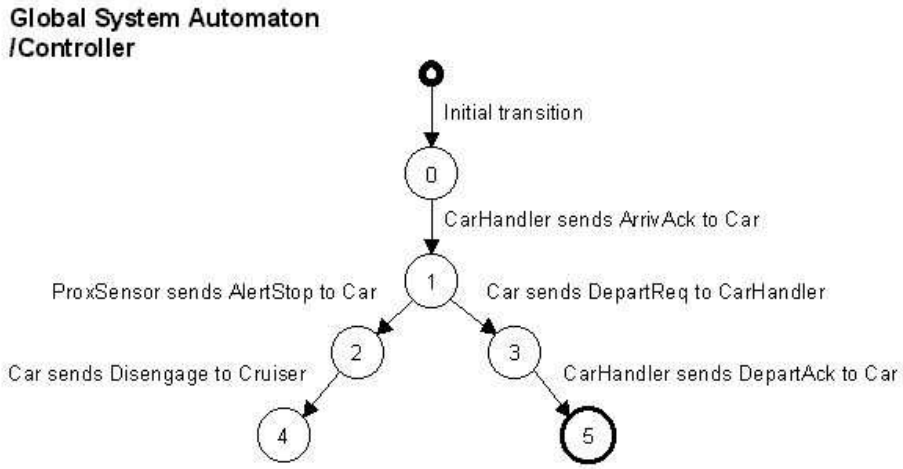

Figure 3.6: A subautomaton (small portion of the GSA) for the railcar controller system is shown. The controller that is synthesized results in an identical state machine. It consists of instructions for all objects in the system to follow, namely, the `CarHandler`, `Car`, and `ProxSensor` objects. The objects have no autonomy. The branching decision (from state 1 to 2 or 3) is made solely by the controller.

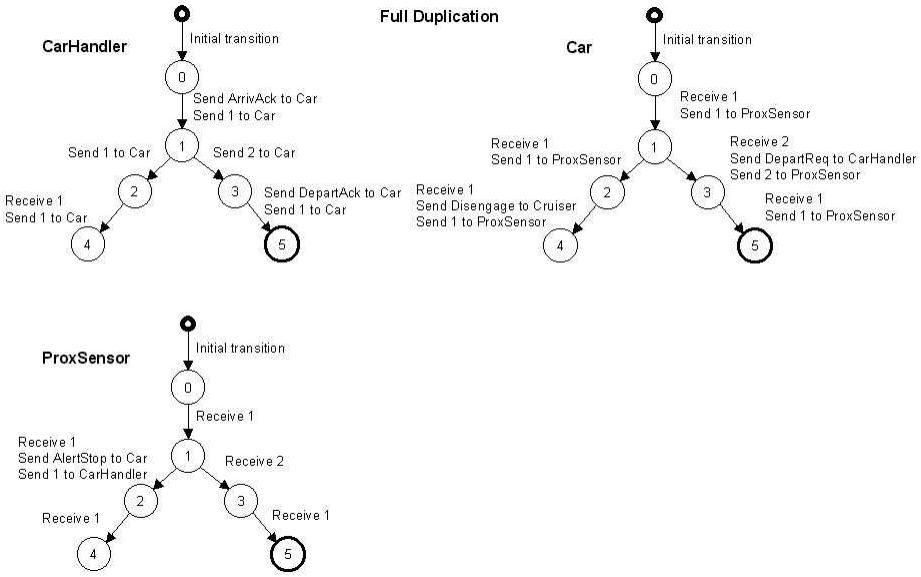

Figure 3.7: The full duplication approach is illustrated here. Each object can send multiple messages on a single transition to any other object. If a message receive is defined for a transition, then the object blocks waiting for it. Otherwise, it proceeds with the send operations. After doing so, the object then sends a *collaboration* message to another arbitrary object in the system so that it completes the same transition, even if it is not involved in the communication. Each object's finite state machine is the same, and the transitions always match across all objects. The collaboration message is forwarded to another object until all of the objects in the system have completed the same transition. The objects orchestrate their transitions without the help of an external controller. The disadvantage is that many collaboration messages are required. Also, the environment is not involved in making branching decisions, which can be useful during simulation. For instance, in the transition from state 0 to 1, `CarHandler` sends the message `ArrivAck` to `Car`, then sends the collaboration message `1`, which is then forwarded from `Car` to `ProxSensor`, so that all objects end up at state 1. `CarHandler` then makes a branching decision either to state 2 or 3. The state symbol with the thick border represents the acceptance state. Again, the state machine shown is not complete.

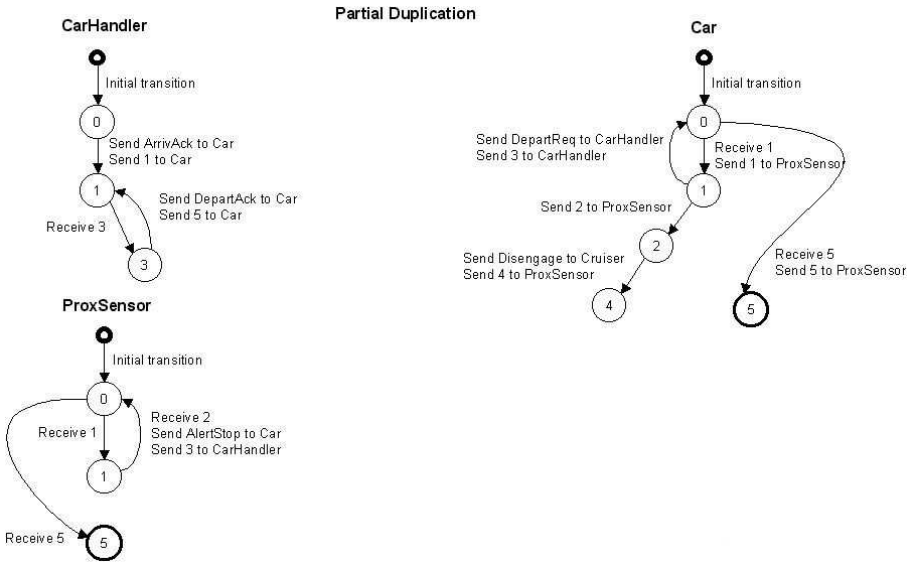

Figure 3.8: The partial duplication approach reduces the state space by eliminating states from an object associated with communication that the object is not specifically involved with (through a send or receive action). For instance, consider the `CarHandler`. States 2 and 4 are only associated with the messages exchanged between `Car`, `ProxSensor`, and `Cruiser`, and so they have been eliminated from `CarHandler's` state machine. Here, `Car` is the object that makes the branching decision, i. e. whether to send the `DepartReq` message, or cause `ProxSensor` to send the `AlertStop` message by sending it the collaboration message labeled `2`.

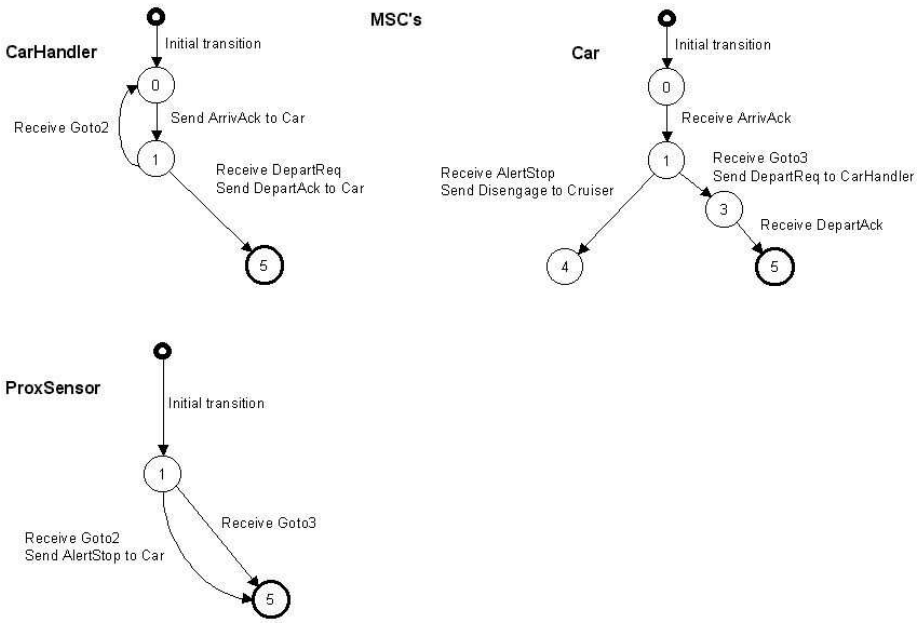

Figure 3.9: The state-based representation of the MSC synthesis discussed in this thesis is most similar to, but differs slightly from, that of the LSC partial duplication result. The state machine shown here is that generated using the maximum progress algorithm (see Section 4.1.2 on page 62). None of the processes make autonomous local or non-local branching decisions (see Section 4.5.2 on page 103), and none of them coordinate with each other using collaboration-like messages. The environment acts like a "controller," but in a lesser capacity, injecting messages to decide branching in the presence of local and non-local choice. For instance, `Car` either receives an injected `Goto3` coordination message at state 1, or it receives an `AlertStop` signal to proceed in another direction.

To support the synthesis of processes with orthogonal components, an additional check must be performed. An object must ensure, before sending a message, that taking a transition will not cause the system to enter a global state where one of the universal charts that make up some of the orthogonal components of a process cannot be executed. Thus, the process must verify that there does not exist a *supercut*, of all of the finite successor supercuts from the current state (where a supercut is the set of cuts through all of the universal charts), that has at least one hot (mandatory) location. Otherwise, taking the transition in one of the orthogonal components will invalidate the run, since that mandatory location will not be visited.

Chapter 4

The Synthesis of ROOM Design Models from MSC Specifications

4.1 Analysis

4.1.1 Approach

In this section, an analysis has been undertaken of the current implementation of the MSC to ROOM synthesis algorithms in the Mesa tool. The usefulness of the synthesis in a realistic programming environment is of principal interest. The expected result of the synthesis is an executable, prototypical model that can be extended, or refined, in the future by programming logic into state transitions.

The system selected for study is a simplified control software for a PBX (Private Branch Exchange) supporting POTS (Plain Old Telephone Service). It is a well-understood system of sufficient complexity to test various aspects of the algorithm. It consists of the following entities:

- The environment, representing the physical interface to the user, in the context of actions initiated, including: the dialing of digits, and sensory feedback, such as auditory dial tones.
- Phone-based call control processes, communicating by asynchronous message-passing. Each phone process has direct access to the hardware control memory, although this interface is left unspecified.

The environment process is similar to any other process in the system, and has no unique characteristics in the ROOM methodology. It is also represented as an actor in the synthesized model.

The high-level MSC view of the PBX system is seen in figure 4.1, and the bMSC's are shown in figure 4.2 [1].

Comparison between automated and hand-coded models

A ROOM model was automatically synthesized from this MSC specification using the maximum progress and traceability algorithms of the Mesa tool. The latter algorithm was chosen for further study due to its greater traceability of actor states to bMSC's, as will be explained. Next, an operational hand-coded model conforming to the same initial set of formal requirements was created from scratch using the ObjecTime Developer tool, imparting knowledge of advanced ROOM constructs.

The finite-state machine behavior of the hand-coded model was found to be similar to that of the synthesized model, but its structure was expanded to take advantage of hierarchy and replication (multiple instantiation from actor class types), and various views were also built. A global, automatic coordinator was also added to automate all branching decisions so that simulation would be viable. These experiments were shown to work through simulation, and the results from the hand-coded model have been illustrated throughout this section, with comparisons made to the results of the

[1] The shapes of the hMSC and connector nodes are slight deviations from the Z.120 standard [55].

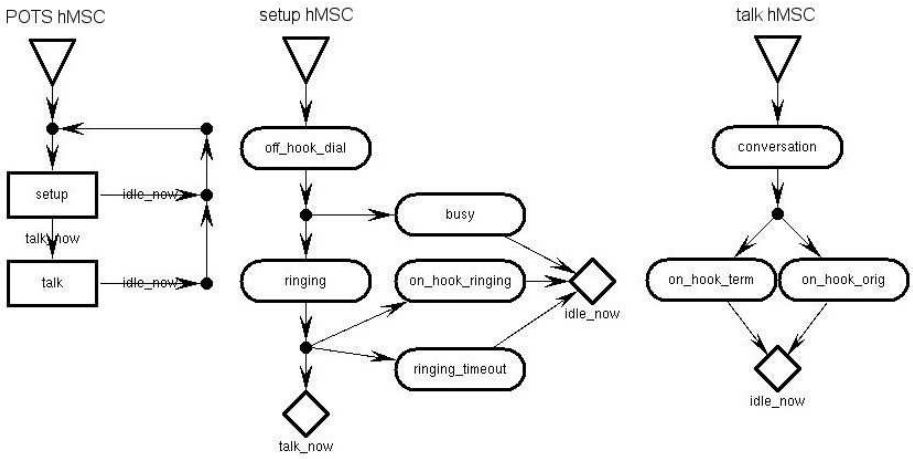

Figure 4.1: The top-level hMSC of the PBX system specification. The topmost hMSC appears leftmost in this diagram, while the setup and talk components are shown to the right of it. The starting point for processing is indicated by the start node symbol ($\bigtriangledown$). Nested hMSC's are represented by the captioned rectangular nodes ($\Box$), while bMSC's are represented by oval nodes ($\bigcirc$). The diamond symbol ($\diamond$) denotes a connection to another hMSC, the label identifying the name of the connector.

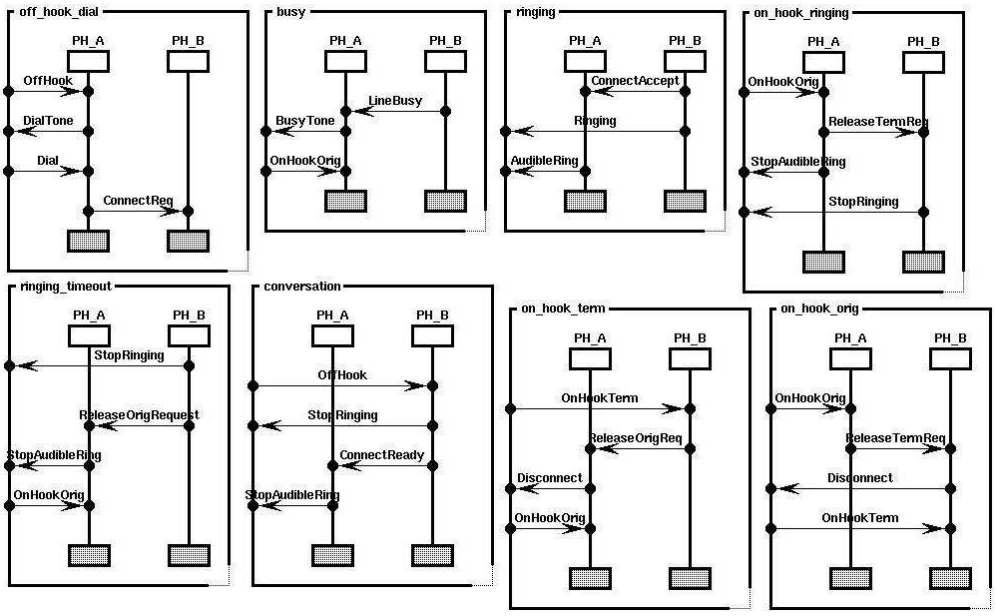

Figure 4.2: The bMSC's making up the PBX system specification.

already implemented synthesis program [5]. Algorithms were then devised to integrate these proposed enhanced synthesis features into the existing implementation of the MESA tool.

4.1.2 Existing synthesis algorithms

This section describes the existing synthesis algorithms implemented in MESA, discussing the structural component of the synthesized model first, followed by its behavioral component.

Structural synthesis

Each concurrent MSC process is instantiated as a component actor of an enclosing system actor. For every pair of distinct processes that is a member of a defined coordination relation (i.e. for which there is a non-empty set of signals exchanged between them), a bidirectional ROOM protocol is defined, corresponding to *in* and *out* signal lists (signals defined in both directions). These coordination relationships, as well as the signal lists, are computed by performing a depth-first search on the underlying hMSC graph, finding all communication partners. Each hMSC node is dereferenced along the way until an underlying bMSC is found, and any new communication events are added to the relation. One of the two actors is arbitrarily chosen as a sender. Each protocol that is defined for a process will result in an end port being generated and bound to the port of the other actor communicated with. A pair of ports thus corresponds to a pair of processes in the coordination relation. If an actor is designated as a receiver, then its port is conjugated, meaning that the in and out signal lists are reversed from its perspective.

Behavioral synthesis

The two algorithms for generating the finite state machines in ROOMcharts that are currently implemented [5] differ with respect to the point at which

transitions are terminated:

Maximum traceability. Each transition in the FSM (finite state machine) of a synthesized actor is terminated whenever a receive event in a bMSC is encountered, and also at the end of each bMSC.

Maximum progress. Each transition is terminated whenever the next receive event is encountered, either in the same bMSC or any successor bMSC. The hMSC graph is traversed to find a reachable bMSC in which a receive event is found. The state machine progresses until the next receive event occurs. The maximum progress algorithm makes use of two types of states:

Wait_for_*x*: Waits for signal x to be received. A trigger for this signal is defined, and a transition is made out of this state once it is received.

CP_for_*x*: A choice point. The signal x is the last to be sent in the bMSC from which branching is to take place. Multiple transitions out of this state are possible, depending on the trigger, i.e. which signal (corresponding to a unique successor bMSC) is received. These signals comprise the triggers.

The key difference is illustrated by examining the activity of the `Environment` process in the `busy` bMSC, as shown in figure 4.2. This event occurs when a call request is made to a phone already engaged in a call. Upon receiving the `BusyTone` signal, the `Environment` process sends `OnHookOrig` to `Phone A`. According to the path in the hMSC of figure 4.1 on page 60, a transition is made to the `off_hook_dial` bMSC, where the first event is the `Environment` sending an `Offhook` message to `Phone A`. The two synthesis algorithms differ with respect to the number of states generated for this single transition:

Maximum traceability. A hierarchical state is synthesized for each of the `busy` and `off_hook_dial` bMSC's, as in figure 4.3. The decomposition

of the states is shown in figures 4.4 and 4.5. The `OnHookOrig` signal is sent in the `OfBu` transition of the `busy` state, in figure 4.4. A timeout signal causes the `BuOf` transition to be triggered. This extra transition is generated because the end of the corresponding `busy` has been reached. The `Offhook` signal is then sent in the `BuOf` entry transition of the `off_hook_dial` state, in figure 4.5.

Maximum progress. The state machine that is generated is flat, as in figure 4.6. The `CP_for_Dial` state signifies that the `Dial` signal has been sent, and a choice point exists at this state. The appropriate choice will be taken depending on the signal received (i. e. to either proceed through the `busy` or the `ringing` bMSC's). The `OnHookOrig` signal is sent in response to the `BusyTone` signal in the `S2S1` transition. The `OffHook` signal is also sent in the *same* transition. Therefore, this algorithm requires one less state to be synthesized.

Thus, in the maximum traceability approach, a ROOMchart is created for every hMSC node (at every level) in the specification, with a hierarchical state defined for every hMSC node), and a ROOMchart is also created for each lowermost level consisting of a bMSC node. A hierarchical state is synthesized for the bMSC node itself. Thus, the hierarchical structure of an hMSC specification is synthesized as a hierarchical state machine in ROOM. Each bMSC is synthesized as a single hierarchical state containing substates

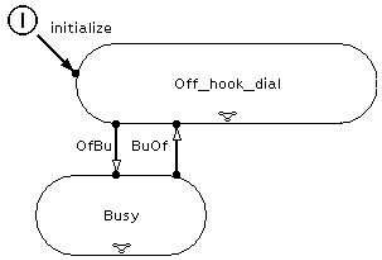

Figure 4.3: The hierarchical state machine of the `Environment` actor generated using the maximum traceability algorithm. The diagram has been simplified to show only the relevant states.

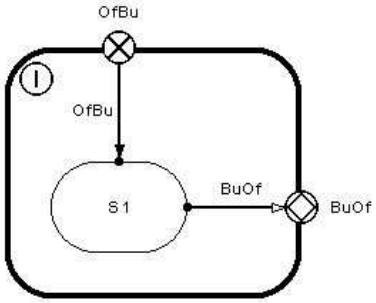

Figure 4.4: The `busy` state machine.

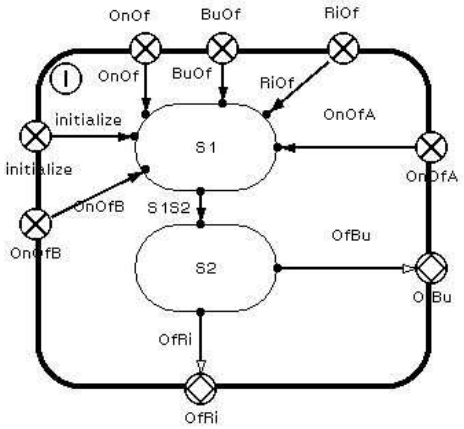

Figure 4.5: The `off_hook_dial` state machine.

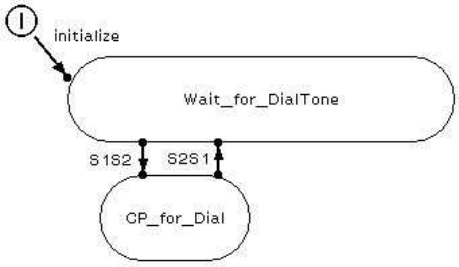

Figure 4.6: The state machine of the `Environment` actor generated using the maximum progress algorithm. This diagram has also been simplified.

corresponding to the local states (send or receive events) of the MSC. This approach results in high requirements traceability, because it is possible to identify in their entirety all of the transitions in a synthesized actor's behavior that correspond to a single bMSC.

This is not the case with the maximum progress algorithm, which results in a smaller total number of states, but at the expense of traceability to the original specification. The structure of each actor will not be identical, and its state machine is flat. Thus, there is a tradeoff between the two algorithms in keeping the effective length of transitions as long as possible to reduce the state space and resultant complexity in the final model, versus introducing more states and hierarchical organization in an effort to improve traceability.

Another option, where a ROOMchart transition would be terminated after every communication event, including sends, was dismissed and not implemented. It would result in many transitions relying on auxiliary triggers to model send events, and result in complex and unnecessarily bloated models.

No actual behaviour is defined for transitions other than reactive signal send events. It is left up to the programmer to program additional logic in RPL or C++, as required, as part of the refinement process in later stages of development. This can include data manipulation, which is not otherwise modeled.

The maximum traceability algorithm is of primary interest in this thesis, for the reasons stated above. The following is the process by which the behavior of actors from MSC's is synthesized:

- The general result is that local control states of bMSC processes, marked by receive events, are mapped to states and enabling conditions for transitions in the ROOMcharts. Any number of signals may be sent through a specified port during a transition from state to state, corresponding to all consecutive send events in a bMSC that follow a control state.

- 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 each top-level state is hierarchically decomposed into states representing the bMSC or hMSC components that make up the main hMSC graph. All connecting edges of the hMSC graph are converted into transitions in the top-level ROOMchart. Therefore, a transition is synthesized for the connection between every bMSC on the control graph, even across hMSC boundaries.

- For every process, a message list is compiled which comprises all communication events that the process participates in in each bMSC. The entries are in the order of the events occurring in the bMSC.

- The algorithm proceeds with the construction of the basic ROOMchart. The first element of the message list is analyzed: if it corresponds to a receive event, then a transition to a local successor state is compiled. This transition includes all entries in the list until either an entry corresponding to a receive event is reached, or the end of the list is reached (and the entire bMSC has been processed).

- If the first message in the bMSC is a send message, then the generic timeout trigger is used in the initial transition, set to the smallest possible time step value [2]. This is essentially flags the bMSC as a target in a branching point requiring coordination from the user in the form of an injected control signal. In the initial transition of the actor, the timeout remains.

- As a result of the above operations, a temporary ordered list of transitions and *next states* is obtained for each ROOMchart. The first

[2]The Simulation Timing Service in OTD defines absolute time in a steady progression of epochs, expressed by a linear progression of monotonically increasing integers. Service methods such as setting a periodic timer are used to request a process to be asynchronously informed at an absolute time or in an interval during a simulation.

element in the list corresponds to the entry transitions into a basic ROOMchart, and copies of this transition are connected with the first state that is encountered in the list. The final states in the list are connected to all outgoing transition points.

4.1.3 Criteria catalogue

The following criteria were identified as being relevant to an analysis of the synthesis from the viewpoint of usefulness to a software developer:

Traceability. The ease of which it is possible to reference the requirements of a design to the original specification, through matching labels rather than deduction. Each component of the model should clearly correspond to a specific aspect of the specification, and for reasons of clarity, should not include more information than is necessary to represent the original requirements completely and faithfully. It is necessary to define an appropriate mapping between these two by-products of development. The use of abstraction is key here. Traceability impacts the ease of debugging a program during simulation, and plays an important role in documentation and architectural recovery in the future.

Understandability. The clarity and conciseness of the design model. The programmer should be able to understand, without unreasonable difficulty, the purpose and meaning of all components, and how they relate to the formal requirements. Strongly mathematical formalisms can restrict their use to a small number of specialists, but a wider audience is desired in general, including C++ applications programmers and technical management. Understandability depends to some extent on *program complexity*. Metrics exist to measure the quantity of various aspects of complexity, including McCabe's Cyclomatic Complexity Metric, discussed in Section 4.1.3.

Maintainability. The degree to which the model is compatible with a testing environment to verify the correctness, completeness, and consis-

tency of the original requirements. Specifically, it should be possible to add monitors to view the inner workings of the model to test its functionality during simulation. Maintainability depends to a large extent on understandability, and requires adequate documentation.

Complexity. The amount of information that the model consists of. It is usually desirable to reduce the state space of the model so that it is possible to store complex and large sets of requirements and designs in memory and run model checking and simulation tools on them. Various metric indices exist, including size (lines of code), interaction with the environment, process complexity, and connectivity. Complexity may also impact on understandability.

Correctness. The design-time model must satisfy all requirements, must be unambiguous in interpretation, and no conflicts can be introduced into the design, assuming none are present in the requirements.

Extensibility. The ease with which it is possible to add artifacts to the model due to new or changed requirements, to modify its architecture, or refine it by programming specific behaviour at each state transition. The integration of code to the model allows more functional prototypes to be built. This is known as *perfective* maintenance. Support for refinement is considered to be the most important criterion in this study.

Program complexity

Measuring the complexity of software is a useful step in determining its understandability and hence the capacity for humans to verify and maintain it effectively.

One of the oldest known measures is McCabe's Cyclomatic Complexity Measure, a measure of the fundamental cycles in connected and undirected graphs with binary decisions. The McCabe number is equal to:

$$M_{undirected} = (e - n + 1)$$

where e is the number of edges, and n is the number of nodes. The McCabe number is significant in that it represents an upper bound on the number of tests needed to ensure that all statements are executed at least once by performing an edge traversal.

On a directed control flow graph, of interest here, the McCabe number is calculated as:

$$M_{directed} = (e - n + 2)$$

and is equal to the number of linearly independent basis paths. The complexity of the two synthesis algorithms described in Section 4.1.2 is the same, as the maximum traceability graph adds a single extra transition (edge) and state (node) for every bMSC termination, as described. For instance, consider two states in the synthesized ROOM model, s_1 and s_2. Suppose that the two states are consecutive, in which case the addition of s_2 to the underlying message graph results in the connection of s_1 to s_2, adding one state and one edge, so that the McCabe number remains constant. If the two states are found in two different but consecutive bMSC's, the maximum traceability algorithm will cause a new state s_3 to be inserted between s_1 and s_2, signifying a transition to a new sub-state (i. e. bMSC). The addition of this state will result in a new edge from s_1 to s_3, and also a new node for s_3 itself. The two will again subtract from each other when calculating the McCabe number, so that it remains the same.

The beneficial modularization characteristic of the maximum traceability algorithm is not reflected in this metric, however. Object-oriented metrics have recently been studied, such as those of Morris [26] and Chidamber & Kemerer [27]. A number of object-oriented characteristics do not appear in the ROOM models being synthesized, and as a result some metrics are irrelevant, including: depth of inheritance, method complexities, the extent of polymorphism, attribute hiding factors, and others. An applicable measure is *application granularity* [26]:

$$\text{application granularity} = \frac{\text{total function points}}{\text{total no. of objects}}$$

where objects would constitute actors and function points would equate to

state transitions in the case of ROOM models. In general, applications constructed from more finely granular objects (a lower number of functions per object) are more easily maintained because the objects are smaller and less complex. The concept of multiple containment discussed in Section 4.2.6 can partition a complex object into multiple interface views, and although the functional complexity remains the same, the granularity decreases.

Another useful measure is coupling between object classes [27], where the definition can be extended to a count of the actors with which an actor communicates by sending or receiving a message to and from them. Excessive coupling is detrimental to modular design, resulting in higher sensitivity to changes, and less reusability. The coupling is a function of the requirements, but the hierarchical structure proposed in Section 4.4.1 on page 89 reduces coupling by containing the interfaces of its sub-actors and only exporting interfaces to outside actors. Any changes to actors are contained within the common hierarchical levels of all bound actors.

In conclusion, the level of abstraction using the proposed structural features of multiple containment and hierarchical structure is increased. As a result, the complexity of the model can be reduced.

4.2 Views

The support for views permits a relation between multiple objects to be represented in more than one way. For instance, consider the original MSC specification as shown in figures 4.1 and 4.2. The specification only presents a one-sided view of call management. Specifically, it only focuses on a single phone initiating a call to another phone, and eventually reaching conversation mode or aborting due to an error condition, such as a busy party at the other end. The specification is restricted to only representing one end of a call, namely, origination. The call scenario that was depicted necessitates that `Phone A` always initiates communication with `Phone B`, rather than being called itself via the same protocol, or making contact with a third party.

It does not include any termination logic so that `phone A` may answer or refuse an incoming call from another phone. Due to a lack of reflexivity, the model represents only a very specific scenario rather than a generic process that can assume the behavior of more than one instance in the MSC specification. Consequently, the two actors synthesized from the two MSC phone instances, as per figure 4.7, are asymmetrical.

The reason for this incomplete specification was to keep the example smaller, but it is a common way of representation whose context is intuitively understood. The engineer assumes that the scenario presented in the MSC is generic and can be mapped to any process instance, although there is no provision for doing so in the synthesized model. The ROOM model that is generated is a very literal translation of each process instance to an actor and ROOMchart, rather than a combination of multiple instances. Hence, only a partial *view* of either phone is modeled — call origination or termination. Each MSC process instance is always mapped to an actor instance of a unique type. As a result, it is difficult to test call management scenarios.

In the ideal situation, multiple instances of an identical phone process type would be synthesized, with complete ability to originate and terminate calls. The situation would then resemble the phone-based actor structure in figure 4.13 on page 79.

Presently, MSC's are used to describe object instance behaviour, rather than class type behaviour. The goal is to combine all possible behaviours of a process into a coherent type definition, as a single finite state machine, that can be mapped onto any instance.

As a solution, the design-time model was enhanced to include both call origination and termination behaviors by merging the two finite state machines, following the algorithm described below. This led to a more satisfactory solution consisting of two replicated self-contained phone processes of the same type definition. The port binding between the two phone processes defines the same protocol in both directions, namely, consisting of identical signal lists.

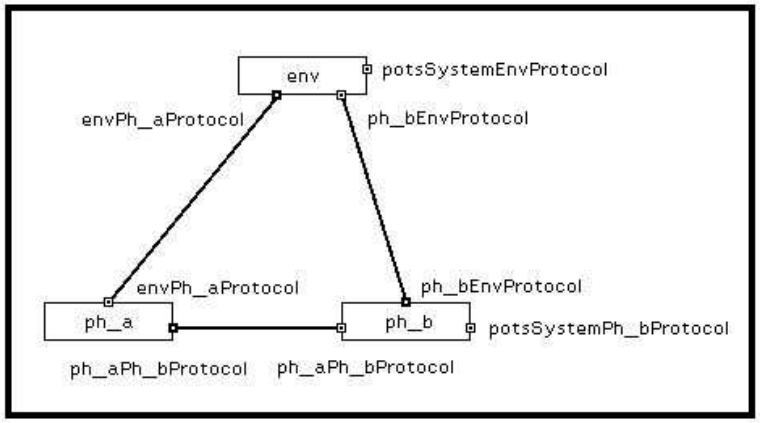

Figure 4.7: Structural diagram from maximum traceability synthesis.

4.2.1 Generalization

The process prescribed involves the integration of behavior of multiple MSC process instances into a single actor type definition. The type may be replicated any number of times on the same hierarchical ROOMchart level. A binding is established between each pair of replicated actors of the same definition, with identical signals being defined in both directions. The protocol is based on the message lists compiled for the communication between the multiple MSC process instances. To merge the finite state machines of multiple actors, a common starting and ending node must be defined. The two may be the same node. The complexity of the algorithm is linear.

4.2.2 Algorithm

1. Let $A = \{A_0, \ldots, A_n\}$ be a set of actors in the system that represent different views of the same process and must be merged.
   Let $B = \{B_0, \ldots, B_n\}$ be the finite state machines of each actor, consisting of hierarchical states, transitions, and merging start and end states.
   Let $ML = \{M_0, \ldots, M_n\}$ be the message lists of each actor.

2. Find the initial node of each actor, which immediately follows the start construct in the MSC specification. Connect the initial nodes of all actors in A with bidirectional transitions. Re-assign all states of the finite state

machines of actors A_1 to A_n to belong to the A_0 actor for behavioral synthesis.

3. Combine the message lists of all actors in A that A_0 communicated with and create protocol P_m. Duplicate each message so that it appears in both the in (incoming) and out (outgoing) lists. Instances of actor class A_0 will be bound together via the P_m protocol.

4.2.3 Example

Consider the MSC process instances `Ph_a`, the call originator, and `Ph_b`, the call terminator. The intent is to merge these two views, namely these two finite state machines, together, into process `Ph_a`. Therefore, A_0 is the actor synthesized from `Ph_a`, and A_1 is from `Ph_b`, which make up the set A. B_0 and B_1 are the finite state machines of these two actors, which make up the set B. M_0 and M_1 are the message lists of these two actors, respectively.

The initial node of each actor in A is state S_0 of the `Off_hook_dial` superstate. The finite state machines of both actors have the same finite state machine on the superstate level, as the maximum traceability algorithm was used. The states S_0 of each actor are connected by the `FSM0` and `FSM1` transitions, as shown in figure 4.8. The programmer must add logic to these transitions so that the correct partial-FSM will be executed. A global coordinator process (described in a later section) can be used for this purpose. The superstate-level behavioral diagram appears in figure 4.9.

The messages that `Ph_a` and `Ph_b` exchanged with each other are combined in the protocol `Ph_aPh_bProtocol` shown in figure 4.10. Two instances of the merged actor class `Ph_a`, representing autonomous phone handlers that can originate or terminate calls, are shown in the structural diagram of figure 4.11, bound via the `Ph_aPh_bProtocol`.

4.2.4 Implementation

The finite-state-machine merging algorithm has been implemented in C++ in the ROOM synthesis engine of the MESA tool. The user specifies, for

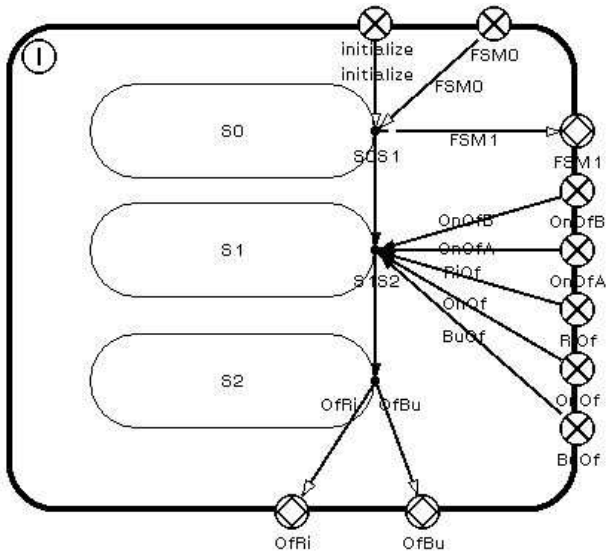

Figure 4.8: The initial node S_0 of the `Ph_a` process, connected to node S_0 of `Ph_b`'s finite state machine, which has been merged with that of `Ph_a`'s.

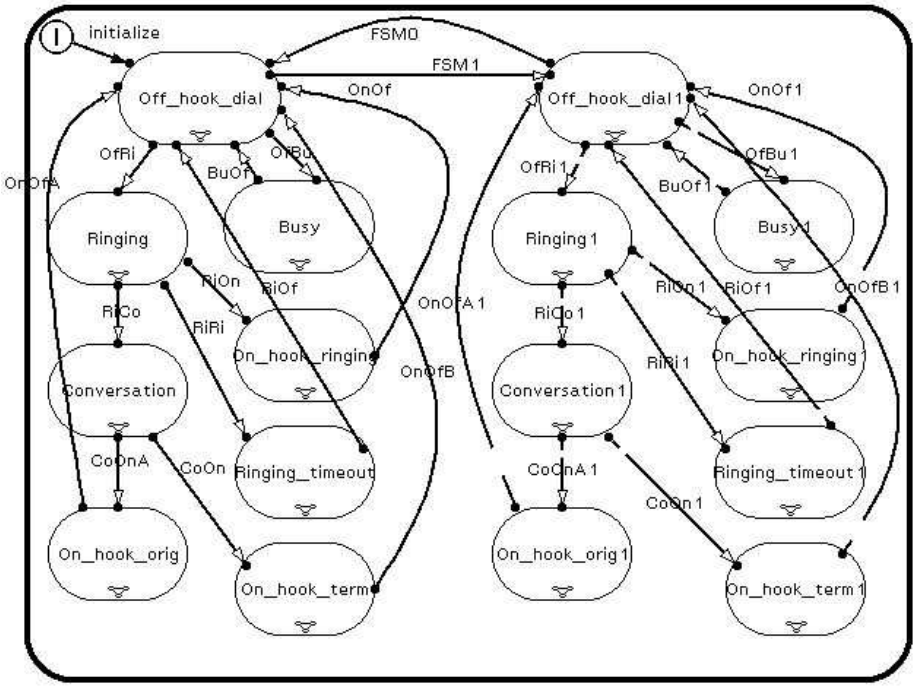

Figure 4.9: The merged superstate-level behavioral diagram of actor `Ph_a`.

In Signals	Data Class
Linebusy	Null
Connectaccept	Null
Connectready	Null
Releaseorigreq	Null
Releaseorigreque:	Null
Connectreq	Null
Releasetermreq	Null

Out Signals	Data Class
Connectreq	Null
Releasetermreq	Null
Linebusy	Null
Connectaccept	Null
Connectready	Null
Releaseorigreq	Null
Releaseorigreque	Null

Figure 4.10: The merged inter-process protocol. The <> symbol indicates that every message is defined bidirectionally.

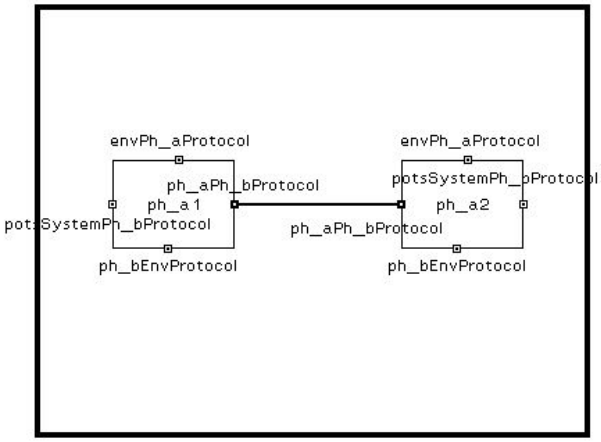

Figure 4.11: Instances of the `Ph_a` actor class bound together via the protocol P_m.

each actor in the system, which actor it should be merged with, by entering the latter's name in the corresponding entry field in the `ROOM synthesis parameters` window, shown in figure 4.12, programmed in Tcl/Tk. The merging function copies all states and transitions of the first actor to the one being merged with, appending a numeric suffix to each name to differentiate between the FSM's. The initial transition of the ROOMchart is not copied, but the first states (connected to the initial node) are linked by two transitions, going in opposite directions, so that the programmer may add logic to switch between the FSM's as appropriate. The signal list of the protocol defined between the original two actors is modified such that each signal appears both in the in-list and the out-list. This enables the merged actor to be multiply instantiated and the instances to be bound together for inter-communication. The synthesized model has been found to compile correctly.

4.2.5 Alternative view

The architectural view described is that of so-called *phone-based* call management. Another possible layout of call control is possible, called the *call-based approach*. Both layouts are used to illustrate new synthesis features in the following sections, such as those of views (in Section 4.2) and hierarchical actor structure (in Section 4.4.1), and therefore must be explained here. The differences are summarized below:

Phone-based. Each phone is statically mapped to a single process (i.e. actor in the ROOM model), integrating both the call origination and termination behaviour as combined finite state machines, as shown in figure 4.13. Each connection involves two communicating phone processes, with each conceptually bound to a unique phone. Only the call origination or termination components are active at one time in each process.

Call-based. Another option is to take advantage of a hierarchical actor structure containing the call origination and termination components

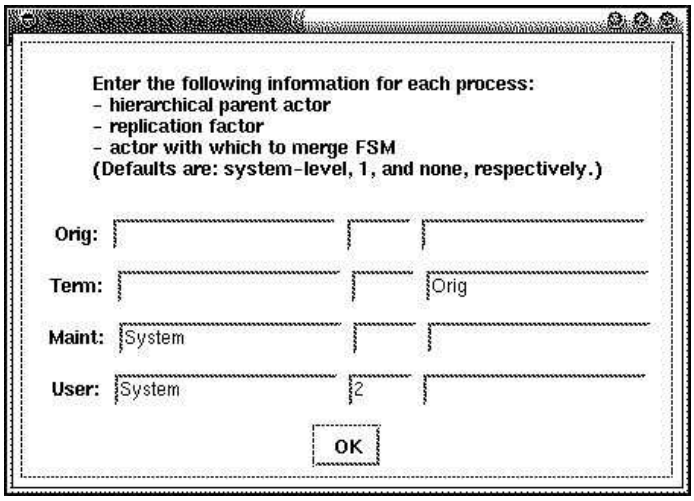

Figure 4.12: The `ROOM synthesis parameters` window in MESA. For each MSC process instance found in the specification, the name of a hierarchical parent actor (that will contain the actor synthesized from the MSC process) can be specified in the adjacent text entry field. A replication factor can also be assigned in the second field. The actor with which the corresponding finite state machine will be merged is indicated in the third field. In the example shown, the `Orig` and `Term` actors (including their structure and behavior) will be merged together into the actor called `Orig`. The `Maint` and `User` actors will become children of a hierarchical parent actor called `System`, while the others will be instantiated on the system-level. Finally, the `User` actor will be replicated twice, while the other actors are instantiated only once.

as communicating sub-actors inside a higher-level process responsible for a single call between any two phones in the system. Once a call is initiated, an available call process responsible for handling both ends of the call, the caller and the callee, is assigned. This concept is illustrated in Fig. 4.14, where the first diagram is the system-level description, while the second shows the interior of the higher-level call actor.

The call-based approach can be implemented using the hierarchical structure feature described in Section 4.4.1, by encapsulating both call origination and termination components within a higher-level actor.

The phone-based approach is more resource-intensive in that it results in at least twice as many processes being active in the system, assuming that each actor maps to a software process in the run-time simulation. It is also redundant in that only the origination or termination component of a phone process plays a role during a call. The call-based approach, however, involves a process with more complicated behaviour and interfaces, and may be more difficult to maintain.

In conclusion, both the phone-based and call-based approaches model the phones as multiple instances of the same type, namely, a self-contained behavioral description of more than one view — in this case, call origination

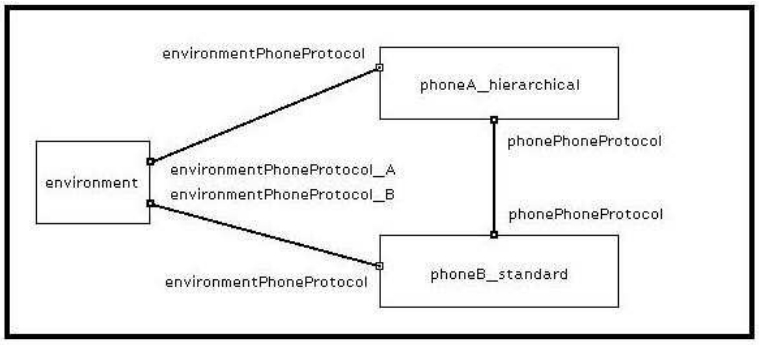

Figure 4.13: Phone-based actor structure.

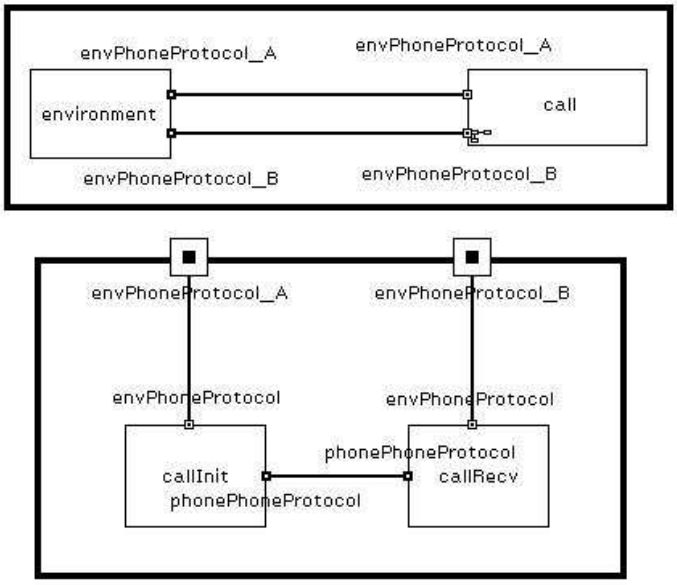

Figure 4.14: The call-based actor structure. The topmost figure is of the system-level actor, defining the bindings of the `call` actor. The figure below it is of the internal structure of the `call` actor itself.

and termination logic. This results in a more accurate and flexible definition of phone processes so that more complicated call processing test scenarios may be simulated. Either view can be built using the proposed finite state machine combination and hierarchical actor structures.

4.2.6 Multiple containment

Abstraction plays an important role in object-oriented systems, by reducing the information and thus complexity of different views of them, in an effort to enhance readability and understandability. Multiple containment is an abstraction feature that allows for the representation of components simultaneously being active in different decompositions. It essentially allows the engineer to specify that two different actor references (to the same class) are bound to the same run-time instance. This is a convenience that simplifies the structural layout of the system by allowing actors to be decomposed into different views called *aspects*. It is possible to view different interfaces of the same actor in different aspects, with irrelevant information in the model (the other interfaces) being omitted.

The idea is based on the concept of an *equivalence* between two or more actor references, in which they all collectively represent the same actor. Each separate appearance of a multiple containment actor is called an *aspect* of that actor.

Multiple containment was not found to be an applicable construct for the multiple view example (figure 4.14) in Section 4.2. Since the same actor must be represented in all decompositions, the `callInit` and `callRecv` (call initiation and termination) components cannot be joined in a new equivalence set in the call-based layout, as they are references to different actor classes. Therefore, it is not possible to view the aspect of `callInit`'s interface to the environment separately from `callRecv`'s interface to the environment.

Multiple containment necessitates that an actor be decomposed according to its defined ports, such that each binding is reflected in a different view, with the composition of all views resulting in a unified actor. An example

appears in figure 4.15.

The use of multiple containment results in a decomposition of an actor according to its communications interfaces, and simplifies the conceptual separation of the different services of a system.

A useful addition would be the support for the merging of independent finite state machines of the same actor reference, each represented in a separate decomposition using multiple containment. This is not possible in the case of call origination and termination behaviors of a call process, because the two behaviors are interrelated. It could be used to good effect in a client-server system supporting multiple, independent services, with each reference defining a partial state machine relevant to the service to reduce its complexity. However, this concept is not realizable in ROOM, because an actor can consist only of a single state machine, and thus cannot be partitioned behavior-wise even across multiple views.

Generalization

Suppose that an actor A_x has multiple bindings to other actors in the system, all contained in the same top-level actor A_{top} (i. e. all references are on the same hierarchical level). Each interface (binding between A_x and another actor) can be individually represented in a different view through multiple containment. Each view will contain a reference to the same actor A_x. This complies with the limitation of multiple containment that no more than one reference to the same actor can appear within the same top-level actor.

Algorithm

1. Let $P = \{P_0, \ldots, P_n\}$ be the set of port references in actor A_x, where each port binds to another actor in the set $A = \{A_0, \ldots, A_n\}$, where A_x and all actors in A are references in the same top-level actor A_{top}.

2. For each binding between A_x and an actor A_y in A, create a sub-actor $A_{y_{view}}$, containing the references A_x, A_y, and their binding.

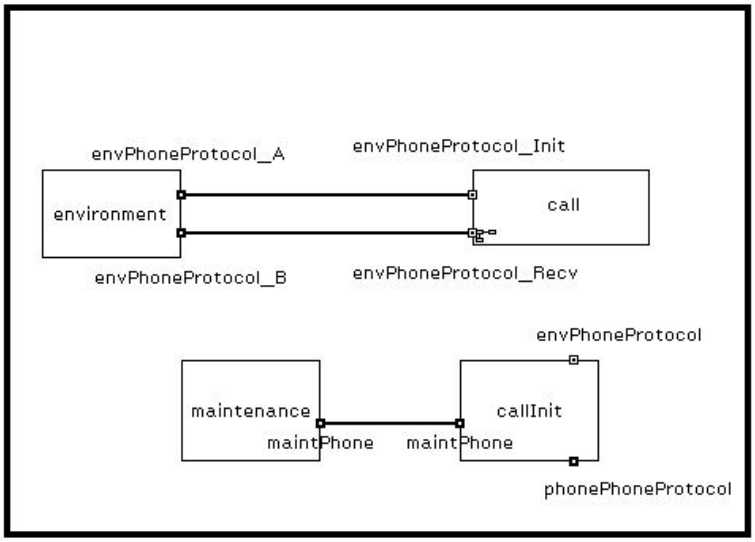

Figure 4.15: The system-level structural diagram is shown. The `CallInit` actor reference is defined within the call-based `call` handler actor, as shown at the bottom of figure 4.14 on page 80. It is also defined on the system-level diagram, as being bound to the `maintenance` actor. This represents automated test and maintenance activity being carried out on the phone when not in use, and represents a system service different from call management. The binding of the actor reference within the `call` actor concerns the communication of switch-hook status and tone signals with the environment, while maintenance control signals are represented in the binding shown on the system-level diagram. Although the `maintenance` process is not relevant to the call processing scenario serving here as the result of the ROOM synthesis work, it does illustrate how multiple containment can be used to display different views of a software system.

3. Replace the references to A_x and the actors in A with references to $A_{y_{view}}$ for all actors A_y in A (the references are not bound).

Example

An example of the application of the above multiple containment algorithm appears in figure 4.16. The `callInit` actor has a number of interfaces, including ports binding it to the `environment` and `callRecv` actors. These two bindings are reproduced in their own sub-actors: the `callInitSystem` and `callInitRecv` actors. The references to `callInit` in the sub-actors are equivalences, and each sub-actor constitutes an aspect. The sub-actors are referenced in a top-level actor, which replaces that of figure 4.7 on page 73.

4.3 Replication

As stated earlier in Section 4.2, MSC's inherently describe object instances rather than class type definitions. The ability to specify types and instantiate them is a useful addition to the work on structural ROOM synthesis in [5]. In the context of ROOM, it comprises translating an MSC instance into an actor class, then instantiating multiple actor references to this class in the structural view.

An example of how this idea can be applied is in the specification of multiple line cards being connected on a shelf and the synthesis of each as a replicated actor, complete with replicated ports and bindings. This allows more complex call management scenarios to be simulated, such as call-forwarding features, involving three or more phones.

Although it is possible to create distinctly-named references to the same actor class, the concept of *replication* is studied here, where a replicated reference is defined as an array of objects implementing the same actor class (with the array being referenced by name and element number in send and receive primitives). Both actors and ports can be replicated, with the multiplicity of bindings being automatically calculated. A requirement that must

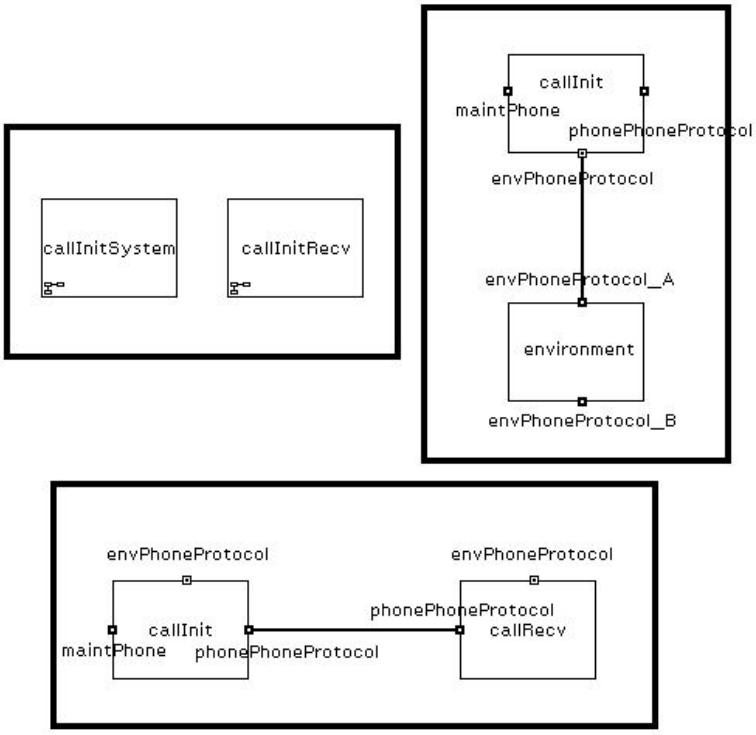

Figure 4.16: An application of the multiple containment algorithm is shown. From top left, going clockwise, the structural diagrams shown are of the top-level actor, the `callInitSystem` actor, and the `callInitRecv` actor.

be satisfied is that the *replication factors* of the actors and ports must be equal on both sides of a binding. The replication factor is the number of times that a specific component is replicated. In other words, only certain configurations are possible such that the number of actors and ports match. Consider the model in figure 4.17 of a call-based system consisting of an environment, and call origination and termination components, labeled `Ph_a` and `Ph_b`[3]. The two call components are each replicated twice, as indicated by the number `2` label on the actor construct. Thus, the system is capable of supporting two simultaneous calls. The same environment actor communicates with both `Ph_a` and `Ph_b`, and hence is instantiated only once. However, each of its ports that binds to the call components must be replicated twice (once per component instance). For example, the `envPh_aProtocol` port is replicated twice, as configured in its property page inside of the OTD tool (not shown). This is known as a *star* configuration, where an actor A_1 is replicated x times on one side of a port binding, while the ports on the actor A_2 on the other side of the binding are replicated x times as well. Thus, the A_2 actor is connected to all of the replicated actors of A_1 in a star-like pattern.

The environment process can choose to communicate with either component by indexing the appropriate port reference in its send function call. To model the interaction between multiple calls, such as a phone attempting to call another party already engaged in an active call, the programmer must add the appropriate call management logic, typically found in a separate call management module. The concern at this point is to synthesize multiple actors from a single MSC process through replication.

4.3.1 Generalization

A replication factor ($\equiv Rf$) is assigned to each MSC process, with a default of $Rf = 1$ being assumed. Syntactically, the factor can be represented by a numeric label next to the MSC process name. Each port on an actor

[3]The coordinator actor shown is a form of synchronization explained in Section 4.5.5.

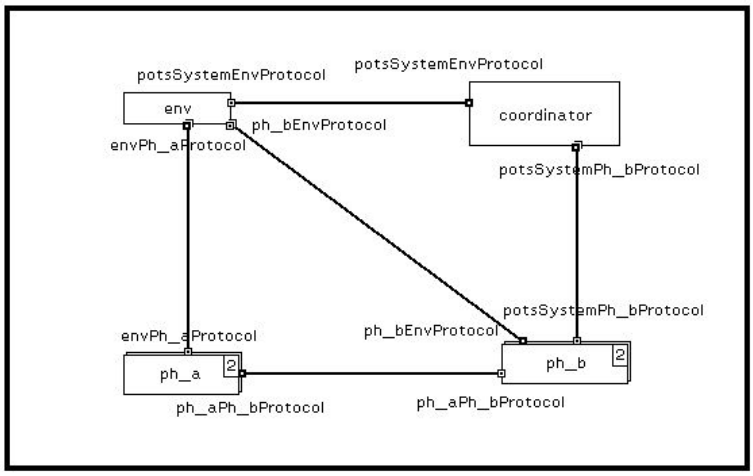

Figure 4.17: Replication of the call origination (`ph_a`) and termination (`ph_b`) components.

whose Rf is 1 is replicated only once. However, every actor A_x bound to an actor A_y that has a higher replication factor, has its ports replicated. If A_y is a hierarchical actor (Section 4.3 on page 84 discusses the concept of *hierarchical actors*), then its relay ports are replicated, and their replication factor is used, instead. In other words, the star configuration is always synthesized.

4.3.2 Algorithm

1. Let $A = A_0, \ldots, A_n$ be the set of actors in the system, each corresponding to an MSC process of the same name.
2. For each actor A_x, if the replication factor $A_{x_{Rf}} > 1$, then replicate A_x by $A_{x_{Rf}}$ times.
3. For each actor A_x in A:
4. For each actor A_y bound to A_x where $x \neq y$, and the binding has not yet been examined:
5. Let $Rf_{diff} = A_{y_{Rf}} - A_{x_{Rf}}$. *Note:* If A_y is a hierarchical actor, then its relay ports were replicated according to the algorithm of Section 4.4.1. Use the replication factor of each relay port as $A_{y_{Rf}}$, in this case.

6. If $Rf_{diff} > 0$ then:

7. Let $Rf_{div} = \frac{A_{y_{Rf}}}{A_{x_{Rf}}}$. Replicate the port of A_x bound to A_y by $\lfloor Rf_{div} \rfloor$ times.

8. If $Rf_{diff} < 0$ then:

9. Let $Rf_{div} = \frac{A_{x_{Rf}}}{A_{y_{Rf}}}$. Replicate the port of A_y bound to A_x by $\lfloor Rf_{div} \rfloor$ times.

Note: If A_x does not divide into A_y exactly (or vice versa), then a replication factor mismatch will occur, and the synthesized model will not be correct (but can be repaired).

4.3.3 Example

Examine figure 4.17 more closely. The actors in the system include `ph_a`, `ph_b`, `env`, and `coordinator`. The replication factors of `ph_a` and `ph_b` are 2. Hence, these actor references have each been replicated twice.

Now, the differences in replication factors between actors bound to each other are compared. Starting with `ph_a` as actor A_x, this actor communicates with `env`, as A_y, which has a replication factor of 1 by default. Rf_{diff}, The difference in replication factors of the two actors is -1. As a result, $Rf_{div} = \frac{2}{1} = 2$. Therefore, `env`'s `envPh_aProtocol` port is replicated 2 times. Next, the difference in replication factors between actors `ph_a` and `ph_b` is compared. Since the difference is zero, no ports are replicated on this binding. The same procedure is followed on the other actors of the system.

4.3.4 Implementation

The replication algorithm specified above has been implemented in C++ in the synthesis engine of the MESA tool. A user interface has been programmed in Tcl/Tk, as shown earlier in figure 4.12 on page 78. The user inputs the replication factor for each actor in the corresponding entry field.

The actors are then replicated, as are their end ports, so that one-to-one bindings are maintained. Message sends use array references to communicate with replicated actors. The replication works in conjunction with the hierarchical structure algorithm, so that the system can consist of hierarchical parent actors and children, and the latter can be replicated. All end ports, including relay ports, are correctly replicated for bindings spanning hierarchical levels. The implementation has been tested and verified through synthesis and simulation of a test model.

4.4 Hierarchy

4.4.1 Hierarchical structure

In the ROOM synthesis work done so far, only a flat structural topography has been modeled. Only actors and their bindings occupying a single structural level are synthesized. The lack of hierarchy defeats the notion of object-orientation to a large extent.

A solution to this is to consider a hierarchical actor structure, in which the structural definition of an actor encapsulates other actors. Multiple processes defined in a bMSC can be encapsulated, and a hierarchical actor containing a sub-actor for each of the MSC processes can be synthesized. Signals communicated by each sub-actor to actors outside of their structural level are transported through relay ports defined on the boundary of the encapsulating actor.

An example of this can be seen in the call-based structure of figure 4.14 on page 80, where the complementary behavioral components consisting of the transitions and message activity of the caller and the callee are each represented within their own sub-actor, with both components encapsulated within a higher-level call handler actor. All of the activity associated with a single phone calling another is contained within this single actor, thus isolating and abstracting this aspect of call management from any other services.

A close examination of the interfaces of the hierarchical actor in figure 4.14 reveals that the same environment-to-phone and phone-to-phone protocols defined earlier are re-used without modification. The ports on the high-level phone actor simply act as non-delaying relays for the sub-actors contained inside. The ports on the sub-actors themselves are regular end ports.

The MSC processes corresponding to the synthesized `CallInit` and `CallRecv` actors can be grouped together in the MSC specification by introducing straightforward structural notation so that both processes would be understood as belonging to the same higher-level actor containment, such as using a hierarchical process instance proposed in [20]. The higher-level encapsulating process appears on the MSC's of the specification labeled with the `decomposed` keyword, and represents the interface to all of its sub-actors. The decomposition appears in its own bMSC, defining all of the sub-process instances, and their communication with outside processes (represented by the enclosing bounding box). An example appears in figure 4.18.

The co-ordination of behavior within the sub-actors requires a control component to be added by a programmer. For example, an on-hook signal from the environment may be destined for either the caller (`callInit`) or callee (`callRecv`) within the higher-level `call` actor (the structural view was previously shown in 4.14 on page 80, which assumes the call-based configuration). The message may be forwarded by including a destination address as a parameter of the message. Otherwise, each sub-actor can communicate with the environment through a distinct port. Messages from the environment can arrive by referencing port addresses. This layout has been synthesized by default in the example.

Representation in MSC's

The modular design representation suggested in [20] is an appropriate form of expressing hierarchical MSC process structure, as illustrated in figure 4.18.

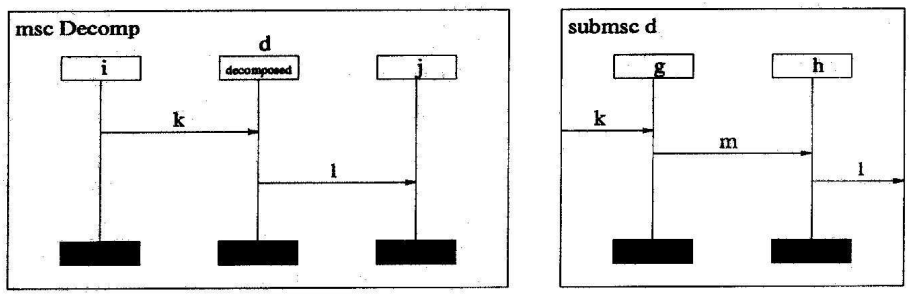

Figure 4.18: The representation of hierarchical structure in MSC's is shown. In the `Decomp` bMSC, the `d` process signifies the encapsulation of subprocesses `g` and `h` in the subMSC of the same name. Messages from outside terminating at the subprocesses are repeated from the environment within the subMSC, such as the message `k`. Messages internal to the subprocesses only appear in the subMSC, such as the message `m`. The `decomposed` keyword denotes the encapsulation [20]. Process instances `g` and `h` would each be synthesized as sub-actors of actor `d`, and bound together.

Generalization

Each hierarchical actor is constructed as a ROOMchart containing its sub-actors. A depth-first search is performed to build the entire hierarchical structure. Actors in the same level are connected by binding their end ports. An actor may also bind to an actor a level above it through a relay port. If the actor is replicated, then the relay port must be replicated with the same factor (Section 4.3 on page 84 discusses the concept of *replication*). A number of assumptions have to be made:

- The same set of processes is defined in each bMSC in the specification. This is a consistency requirement that is verified by MESA.
- Each actor can contain any number of actors to any level of depth[4].
- Each actor requires a unique bidirectional communications channel to every other actor at every level that it communicates with. All

[4]An unspecified physical limit probably exists in OTD.

synthesized ports must be bound. An actor must communicate with another actor in the same level or at most one level above it.

Algorithm

The following is a process for the construction of a hierarchical ROOM model, extended from the flat-structure algorithm of [5]:

1. *Definitions:* A_s is the set of actors $\{A_0, \ldots, A_n\}$ contained in the system-level (highest) actor. Each actor A_x in A_s is defined as a tuple $\{A, Ports, P, Bind\}$ with the following definitions:
   A is the set of actors contained in A_x through encapsulation.
   $Ports : endports \longrightarrow conj$ where endports is the set of end ports and $conj = \{0, 1\}$ is the type of port: ordinary (0) or conjugated (1).
   P is the 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 between ports.

2. *Inputs:* Partial MSC specification B, consisting of a finite set of bMSC's $N = \{N_0, \ldots, N_n\}$, as defined in [5]. Each element N_x in N is now a set of bMSC nodes contained in N_x, such that N is recursively defined. By default, a set is empty unless otherwise defined.
   Outputs: The hierarchical structural components of the ROOM model $R = \{A, Ports, P, Bind\}$.

3. Create the system actor A_s. Recursively descend N, and for each process in every bMSC node N_x, add an actor with the same name to the actor set A_x.

4. Instantiate the highest-level actors (elements of N) in the system actor A_s. For each new actor, create a new structural ROOMchart and populate it with its component sub-actors (populate bMSC corresponding to N_x in N with elements of N_x) by performing a depth-first search.

5. For each actor contained A_x in A (i.e. elements of A):

6. Instantiate it in that actor, A.

7. Create a list of messages L for A_x (composed from all of its contained actors). For each message M_i in L:

8. Create a new protocol $Prot$ corresponding to the *in* and *out* messages of the source and destination processes of M_i.

9. If($Prot$ or $conjugate(Prot)$) $\notin P$, add $Prot$ to P in A_x.

10. 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.

11. Else, add M_i to $InSig$ list of P.

12. Call procedure CreateBindings(Actor A_s).

13. Procedure *CreateBindings*(in parameter Actor A_x)
{

14. If no actors are contained in A_x, create an end port for every other process (within the same ROOMchart) that A_x communicates with based on L, then exit procedure.

15. For each actor A_y in A_x:

16. Call procedure CreateBindings(A_x).

17. Create relay port $Port_y$ for actor A_y contained in A_x, where the actor that A_y is communicating with is not in the same ROOMchart. If A_y has a replication factor of $A_{y_{Rf}}$ (see Section 4.3 on page 84, then replicate the relay port $A_{y_{Rf}}$ times.

18. For each message M_i in the message list L:

19. Let $A_{src} = A_y$ be the source actor and A_dst be the destination actor.

20. If A_{src} is *not* contained in the same actor as A_{dst}, then we assume that A_dst is contained on a higher level. Instantiate relay port $Port_{srcdst}$, add it to $Ports$ of A_x, and bind it to A_{src}'s end port. $Port_{srcdst}$ is then bound to A_{dst}'s end port in A_x's parent using protocol $Prot_{srcdst}$.

21. If both A_{src} and A_{dst} are contained in A_x, then bind their existing ports (end or relay) using $Prot_{srcdst}$.

Example

Consider figure 4.19, reproduced here for convenience. Suppose that the `call` actor is composed of the `callInit` and `callRecv` actors, as shown. Through a traversal of the bMSC's in the partial MSC specification, four process instance definitions are found, including: `environment`, `call`, `callInit`

and `callRecv`. The highest-level actors are instantiated in the system actor, namely the `environment` and `call` processes. Next, a depth-first search is carried out, and the `call` actor is found to contain sub-actors itself. A structural ROOMchart is created for this actor, and the `callInit` and `callRecv` sub-components are instantiated inside of it.

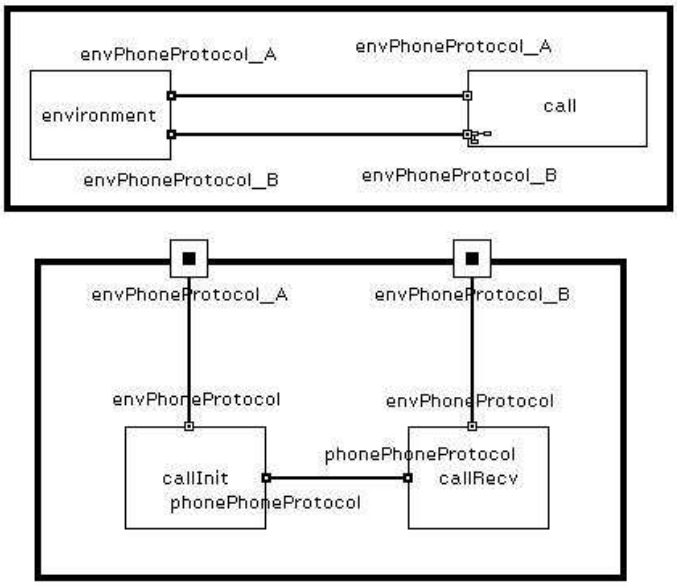

Figure 4.19: Example of the application of the hierarchical actor structure algorithm. The system-level actor appears topmost, while the internal structure of the `call` actor is shown below it.

For each actor and sub-actor in the system, a list of messages is compiled based on send and receive actions. For every message exchanged between two actors, a protocol is created and added to the master list of protocols. Every message in the message list is added to either the incoming or the outgoing signal list of the corresponding protocol. For instance, `Ph_a` sends the `ConnectReq` message to `Ph_b` in the `off_hook_dial` bMSC. Therefore, the `ConnectReq` message will be appended to the outgoing list of the `phonePhoneProtocol` protocol.

The next step in the synthesis process entails the ports and bindings

for the protocols defined between pairs of actors. Starting with the system-level actor, every actor contained inside is traversed recursively by calling the *CreateBindings* procedure. Suppose that the `call` actor is selected, first. It contains two sub-actors, so that the procedure is called on one of them: the `callInit` actor. As it contains no sub-actors itself, an end port is created for the `phonePhoneProtocol`, to be bound later to its companion `callRecv` actor. Also, an end port is created for communication with the `environment` actor via `envPhoneProtocol_A`, even though it is declared outside of the `call` actor. Likewise, two end ports are synthesized for the same two protocols for the `callRecv` actor. By this point, all sub-actors of the `call` actor have been traversed, and all end ports for communication have been synthesized, although the bindings have not yet been.

Now, it is necessary to create the bindings to actors outside of the scope of the current `call` actor. For example, the `callInit` actor communicates with the `environment` via the `envPhoneProtocol`. Therefore, a relay port is created for this protocol, with a unique name, and the relay port is bound to the end port of `callInit`. Likewise, `callRecv`'s interface to the `environment` actor is synthesized through a relay port and binding. As the last step, the procedure requires that bindings between all sub-actors instantiated in the same process be completed by connecting the existing end ports. In this case, `callInit` is bound to `callRecv`.

Once the recursive procedure call returns to the system-level actor, the final bindings in the system take place, between the existing end ports of the `environment` and `call` actors. The hierarchical structure of the system has now been built.

Implementation

The hierarchical actor structural synthesis algorithm has been implemented in C++ in the synthesis engine of the Mesa tool. A user interface has been programmed in Tcl/Tk, as previously shown in figure 4.12 on page 78. The user enters the name of the hierarchical parent of each MSC process instance

that is found in the specification. A hierarchical actor is then synthesized for each parent, with the children actors contained inside of it. The relay ports and bindings (connecting the children to actors outside of the parent) are automatically synthesized, and conjugated as necessary. The bindings between children of the same hierarchical parent actor are also synthesized. Up to two levels of nesting depth are supported. The implementation has been tested and verified on a test model through simulation.

Hierarchical layers

An alternative to hierarchical structure attained through the use of actor encapsulation and relay ports is the use of layers for "vertical" communication. SAP's (Service Access Points) and SPP's (Service Provision Points) are defined for layers by creating references to communication service protocol classes. SAP's can be viewed as the interfaces of the users of a service, while SPP's are the interfaces for the providers of the service. SAP and SPP access points are defined for individual actors in the system, and are bound together.

While ports are bound by connecting a binding construct between two end or relay port references, layer SAP's are bound by reference name alone. A layer SAP is automatically bound at run-time to the layer SPP that shares the same reference name. All SAP's are bound to the first SPP that registered for binding under that name. Both access point types can be registered and deregistered dynamically, with binding or unbinding taking effect immediately based on the existence of references defined in the system. More than one SPP cannot register under the same name. Unfortunately, OTD does not have the capacity to visually depict layers and their bindings, and so their existence is not evident in the structural diagram of a model.

The layering feature was found to work correctly in the telephony model by replacing end ports and their bindings with SAP and SPP references in the transition codes. All protocols are compatible with either mechanism.

Layers are primarily useful when a shared service is accessed by a very

large number of clients, or when shared functionality is of such secondary significance in a system that it is useful to abstract it out of view [12].

4.4.2 Hierarchical behavior

Consider the FSM of the phone actor model generated by the maximum progress algorithm of MESA, as shown in figure 4.20. All of the states are synthesized on a flat topology. There is no conceptual grouping present, resulting in a potentially large state space that is difficult to analyze by a programmer. It is also unclear as to which states are derived from which bMSC.

A solution is to group states in a hierarchical fashion. The presence of a hierarchy in the state-based representation of actor behaviour increases the traceability of the model to the original MSC specification. Each bMSC node can be mapped to a single encapsulating *superstate*, with the result being an easily understood multilevel state diagram due to better abstraction. The hand-coded version with hierarchical states is shown in figure 4.21. It is essentially the same as that generated using the maximum traceability algorithm of [5].

Inside each state corresponding to a bMSC node are the local states of each message receive junction in the bMSC. For instance, the behavioral description of the `off_hook_dial` state is illustrated in figure 4.22. The initial point is indicated by the circled `I` symbol, which indicates a transition into the the first substate upon entry into the enclosing superstate. At the exit point of the bMSC's, a choice is made as to which succeeding superstate is to be entered next. A history is maintained for a superstate such that the current substate is always remembered whenever execution returns to the superstate. The history is erased on an exit from the superstate, however.

Hierarchical states do function in the same manner as a flat-based state machine, with the exception of incoming high-priority messages that interrupt processing and affect the history feature of the substate. The assump-

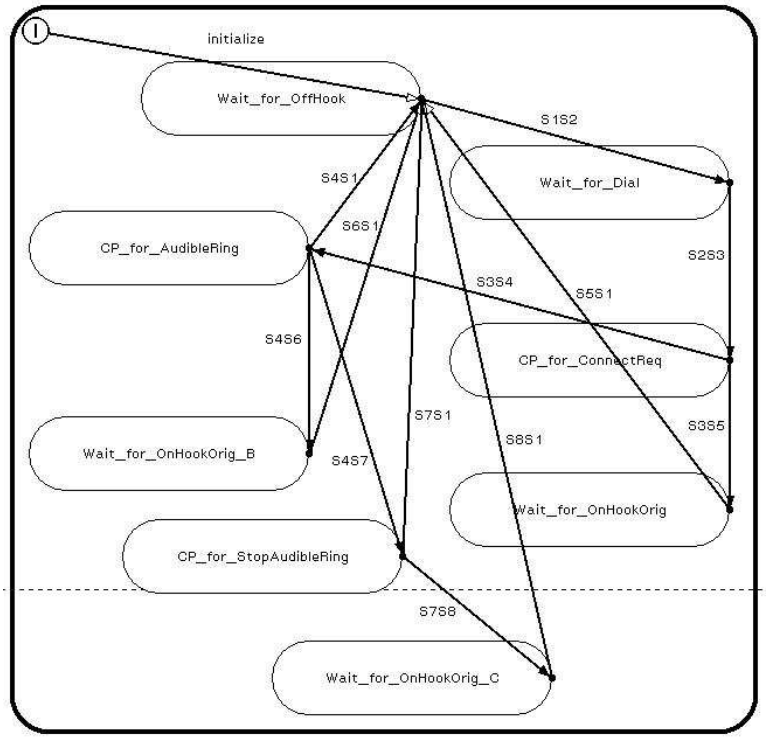

Figure 4.20: Behaviour of the `Phone A` actor generated in maximum progress synthesis.

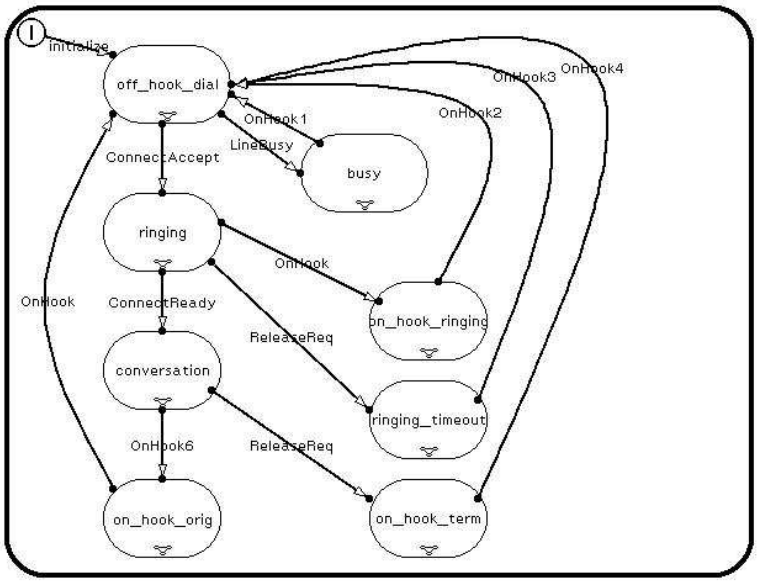

Figure 4.21: Hand-coded behavioral description of the call-based `Phone` actor using hierarchical states.

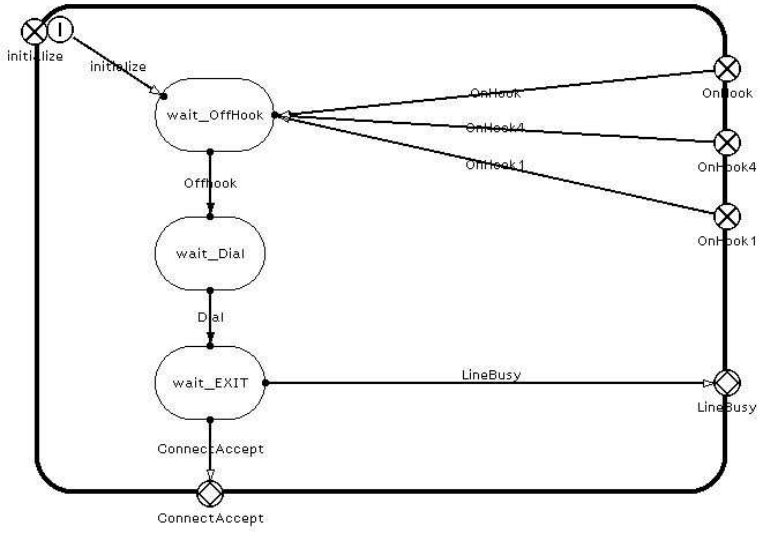

Figure 4.22: The internals of the `off_hook_dial` superstate of the `Phone` actor.

tion in this text, however, is that all messages exchanged within the system are of equal priority.

Data in the model is also accessed hierarchically. Each state represents a lexical scope, and the scoping rule for a state follows a nested approach for any defined variables [11].

4.5 Non-determinism

4.5.1 Transition model

A number of decisions must be made throughout the execution of the behaviour of an actor. Specifically, a non-deterministic choice must be made as to which one of multiple branches is to be taken when exiting a superstate. Each transition from the exiting substate must be bound to a specific transition in the higher-level state machine, and so the choice must be made within the substate logic itself. To do this, it is necessary to include the first message reception, or trigger, of each succeeding bMSC, then to add an exiting transition to the corresponding successor substate. Any messages that are sent in response to this received message can be included in the transition code of the exiting transition of the substate, or of the entering transition of the succeeding bMSC. Although both techniques were confirmed to work identically, it will be shown that the latter is preferred for readability reasons.

Locality of transitions

For maximum traceability, all message events of a bMSC should be synthesized within the scope of execution of the corresponding superstate of the actor's FSM. In other words, assuming the use of the maximum traceability algorithm, each send and receive event of a bMSC would ideally be synthesized inside of the finite state machine of the appropriate actor in the superstate corresponding to that bMSC. Therefore, any send or receive

event that is synthesized in the actor's superstate should be traceable to an event in the corresponding bMSC, and not any other.

One difficulty, however, lies in the transition between superstates. Consider the `off_hook_dial` state of the `Phone A` actor shown in figure 4.22, corresponding to the `off_hook_dial` bMSC. Two possible exits from this state are possible, the destination being either the `ringing` or the `busy` state (corresponding to the bMSC's of the same name). An enabling event is defined for each of these transitions. For instance, the transition to the `ringing` state is programmed to occur on the reception of a `ConnectAccept` signal from the phone being dialed (the `ConnectAccept` signal is the first message event in the `Ringing` bMSC). Next, any send action that must occur as the result of receiving the triggering event is specified in the transition code of the state being entered. In this case, the receipt of the `ConnectAccept` trigger results in an `AudibleRing` signal being sent to the environment, i.e. upon reception of the `ConnectAccept` message in the `Ringing` bMSC, phone A sends the `AudibleRing` message to the environment. This action is programmed in the entry code of the `ringing` state, even though the trigger is actually received in the previous `off_hook_dial` state. This inclusion of the sending action in the state which is being entered results in a truer translation of the MSC specification. As many messages as possible are contained in the superstates that correspond to their original bMSC nodes.

Relating this concept to bMSC's, the message events defined in each bMSC are mapped to a ROOMcharts superstate of the same name. The only exception is that the first message events of successor bMSC's are moved up to the state corresponding to the branching bMSC node, so that they act as triggers in branching decisions. Any send events run in response to these triggers are still localized in the successor states for higher clarity. In other words, transitions are localized for greater traceability.

Multiple entry and exit transitions

Multiple transitions into and out of a state require unambiguous naming for correct resolution. In the interest of clarity, the name of each transition is the same as that of the message that enables (i.e. triggers) it. The names of states from which only one exit is possible are labeled `wait_X`, where `X` is the signal that will trigger a transition from out of the state. The one exception is `wait_EXIT`, a state from which multiple transitions exiting the substate can be taken. In the example of figure 4.22 on page 99, two exit triggers are defined: `ConnectAccept` and `LineBusy`.

Due to these necessary exit transitions, there is admittedly some overlap between signals from the viewpoint of the original bMSC's in which they are defined, as explained in the previous section. It is minimal, however, and only one new state is introduced to handle these multiple exits from the substate, namely, the `wait_EXIT` state. Following the example, the `ConnectAccept` trigger must be defined in the `off_hook_dial` state so that the correct transition is taken, even though it originally belonged to the `Ringing` bMSC.

Although arrived at independently, the maximum traceability synthesis algorithms were found to already generate a transition model similar to the one found to be most intuitive in the hand-coded design process, with the exception of the unnecessary generic state names inside substates, which are labeled from S_1 to S_n, where n is the last state created. The state names in the handcoded version are called `wait_X`, where `X` is the name of the expected trigger.

A restriction of ROOM is that every transition of the FSM graph that enters the same state must have a unique name, regardless of which signal enables it. Thus, duplicate transition labels need to be differentiated using an automatically incremented numeric suffix.

It is possible for multiple transitions to exit from a state, and have some or all of them be enabled by the arrival of the same message. ROOMcharts

specify that the choice is non-deterministic. However, only the signal that is "found first" by the ObjecTime tool is executed, while the others are ignored. The choice is compiler-dependent, and therefore not truly deterministic [12].

4.5.2 Non-local choice

MSC specifications in general are prone to the hazard of non-local choice, in which a unique branching choice may not exist [59]. This is due to underspecification — although syntactically legal, it must be resolved before implementation. The syntactic detection of non-local choice, and its handling through synchronization methods described below, relieves a designer from the burden of explicitly coordinating branching early in the design stage. Non-local choice can eventually be resolved through additional messages making up a coordination protocol [59].

For example, consider the termination of a call. In the environment of the system being modeled, the process representing the environment encompasses the activities of the two human users of the system. As specified in the `on_hook_term` bMSC in figure 4.2 on page 61, if the user of phone A wishes to disconnect, then the `Phone A` process receives an `OnHookOrig` signal. It then requests call termination by sending a `ReleaseTermReq` signal to `Phone B`. However, if the user of phone B disconnects first, then the `Environment` process sends an `OnHookTerm` signal to `Phone B`, which in turn responds by sending `ReleaseOrigReq` to `Phone A`. In either case, the `Phone A` process uses a *wait-and-see* strategy [59, 65] to determine its course of action. The applicability of the wait-and-see approach in a branching situation determines the classification of the branching [59, 65]:

Local branching choice. The wait-and-see strategy can be used to resolve non-determinism within each process.

Non-local branching choice. The wait-and-see strategy *cannot* be used. Explicit synchronization between processes is required.

A required normalization is to ensure that for each branching bMSC node, the successor bMSC's do not share a common prefix of message exchange sequences. Any such prefix is delegated to the end of the branching bMSC node. This does not affect the behavior of the system, but simplifies the detection of non-local choice. A solution to non-local choice is the use of synchronizing history variables, discussed below.

4.5.3 Synchronizing history variables

Synchronizing history variables [40, 59] enable the model to continue functioning even in the presence of process divergence, where a lack of synchronization can cause a process to have its message channel flooded by other processes, regardless of its length, before it has the opportunity to respond. The use of history variables ensures that processes cannot diverge in execution in more steps than the variables' lengths allow. In addition, history variables are useful for resolving non-local choice situations by having a process make a branching decision that the rest of the system must follow, depending on which process is furthest ahead in execution.

Essentially, for each non-local choice branching point, a historical record is kept of the last branching decisions made at that point up to a finite depth. The length of a history variable (usually implemented as an array) determines the number of prior decisions that are kept in memory. Processes synchronize through a collection of global counters, which keep track of how far each process is ahead in execution. Each process, once it reaches one of the branching points, either makes a new branching decision and records it if no other processes are ahead of it in execution, or else follows the same branch taken by another process that arrived at the same point earlier in time. Each branch taken is recorded in the branching point's history variable.

The proposed coordination system for automatic model simulation that will follow in Section 4.5.5 is a different concept, and a comparison of the two will be made once it is explained.

4.5.4 Choice points

A number of alternatives were considered for the implementation of branching in actor FSM's. One of the possibilities is to replace the described triggers with *choice points*, special constructs in ROOMcharts. Through their use, one out of several enabled transitions can be taken based on the Boolean result of the evaluation of a logical predicate. However, such choices are inherently deterministic. In addition, only binary (true or false) decisions can be made, and so there is no support for two transitions exiting a state using a single choice point. Non-determinism can be introduced through the use of a random number generator function, and a choice point can be added for each possible transition out of a state, similar to an if/else-if/else construct in procedural programming languages such as C, but this is an awkward solution requiring the introduction of many states. As stated in Section 4.1.3 on page 68, an increase of the state space of the model is undesirable due to greater complexity. Thus, choice points were not found to confer any useful advantage over message triggers.

4.5.5 Automatic coordination

The original method of decision-making in choosing a successor from a branching bMSC node in [5] relies on the injection of control messages, during simulation, into the auxiliary ports of the actors corresponding to the branching nodes. For example, consider the case of the `ringing` state in figure 4.1 on page 60. The successor states include `on_hook_ringing`, `ringing_timeout`, `busy`, and `conversation` (corresponding to the bMSC's of the same names). An appropriate branching choice at this point requires the user, during simulation of the model, to attach a probe to the end port on `Phone B` bound to the system-level actor, and inject a control message to trigger a transition into the appropriate successor superstate. For instance, the `Doofbu` signal will trigger a transition into the `busy` state. An example of the injection of this message during simulation is shown in figure 4.23. The injection port itself is unbound. This process of injection must be re-

peated every time that a branching decision at that point must be made, and this can seriously impact on the time that a user must spend performing simulations.

The synthesis of an autonomous global co-ordination mechanism would enable these choices to be made automatically, without requiring user intervention during simulation, as is the case with [5]. Breakpoints in the form of port dæmons halting on specific transitions may always be set up local to a problem area whenever debugging is required, but the problem of manually injecting a message for every single trigger point encountered during execution, as in [5], is alleviated. An entity called the global controller replaces the user in sending control messages to participating actors at branching points. Although the use of an automatic coordinator does not add or modify any information in the system model in terms of the original MSC specification itself, it is a useful design-time aid in executing the model that is synthesized. The operation of the automatic coordinator is detailed in the example below.

Mechanism

A single system-level coordinator, manifested as an actor called `Coordinator`, is responsible for making non-deterministic choices at branching points. It interfaces with all actors in the system that require branching choices to be made, as shown in the structural layout in figure 4.24.

In this example, the `Environment` actor, during the transition into substate `S2` of the `Ringing` state from which the branching must take place, sends a request to the `Coordinator` to arbitrate the execution flow. The behavioral description of this state is found in figure 4.25. The signal sent is `Askri`, the "`ri`" suffix derived from the current `Ringing` state.

Inside of the `Coordinator` actor, the coordination request enables a transition out of the `Idle` state, as shown in figure 4.26, and causes a response to be sent to the `Environment` actor in the form of a signal indicating which branch is to be taken. The `Coordinator` is aware of all possible alternatives,

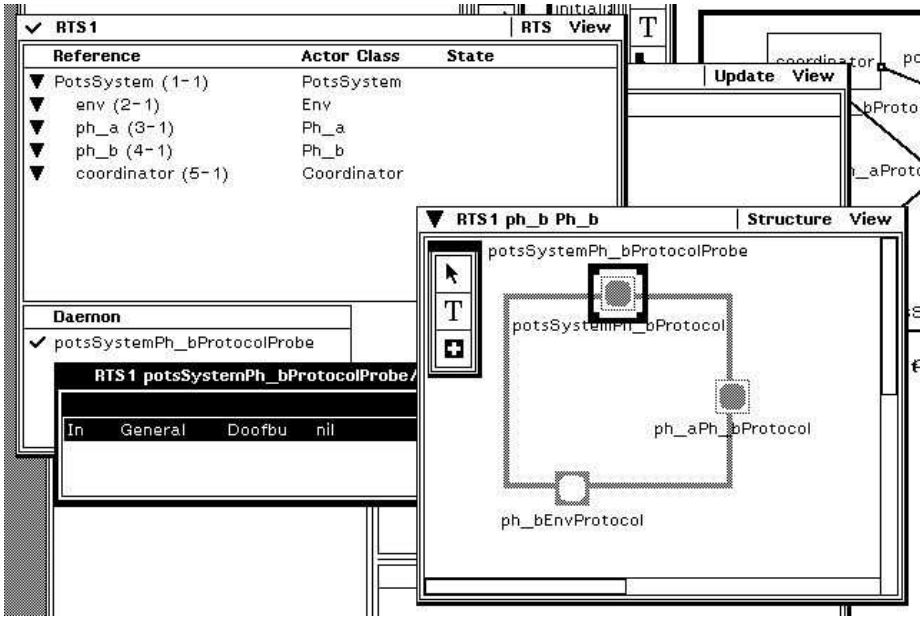

Figure 4.23: An example of the injection of the `Doofbu` message into the probe attached to the `potsSystemPh_bProtocol` port, on actor `Ph_b`, is shown. The list of injected messages appears on the left, while the probe dæmon is indicated on structural diagram to the right by the square construct overlaying the port symbol.

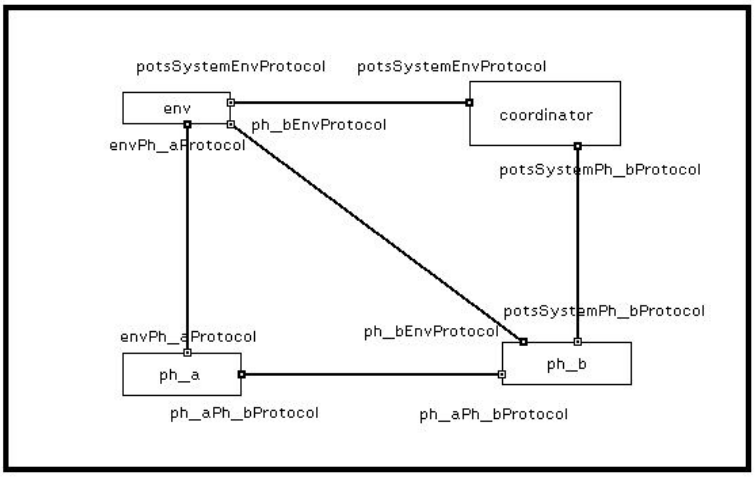

Figure 4.24: Interface between the coordinator actor and the rest of the system.

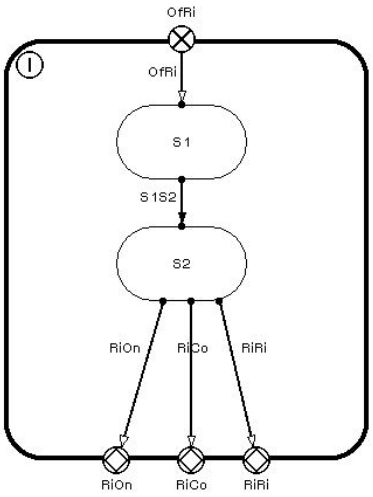

Figure 4.25: Coordination required inside of the `Ringing` state of the `Environment` actor.

and the choice is made non-deterministically on the basis of random number generation. The transition code is found in figure 4.27[5].

The response sent by the coordinator is in the form of a `DoX` message, where `X` is an acronym for the successor state chosen. For instance, the choice to proceed to the `ringing_on_hook` state is executed as the result of the `Environment` actor receiving a `Dori` signal (the "`ri`" signifying `Ringing`) from the `Coordinator` actor.

Multiple entry points

It is possible for more than one transition into a state to be programmed with the same coordination request, of the same name. This occurs if at least one entry from another substate, or possibly the initial transition of the system, is made into the substate where a branching choice must be made, in addition to the standard internal transition from a previous substate. In other words, this occurs when the indegree of a substate is higher than one,

[5] Timeouts are not presently supported in ROOM model synthesis. Thus, the `StopRinging` signal sent by `Phone B` represents the point at which ringing has stopped without the phone being answered. The decision to proceed with this choice is made immediately. As it is directly relevant to the (in)action of the environment, an extra message called `NoActivity` was inserted at the beginning of the `RingingTimeout` bMSC, sent from the `Environment` to `Phone B`. Thus, the environment solely determines which one of the three possible branchings is taken, and the choice is *local* in this case.

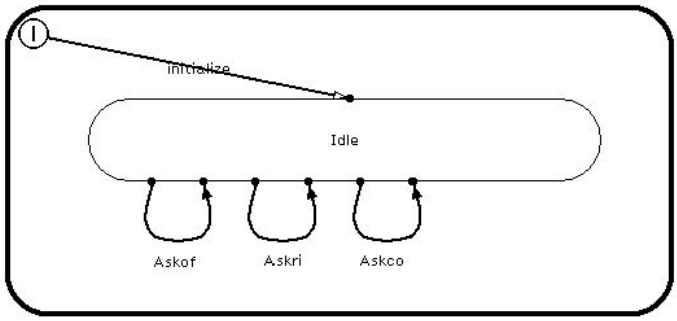

Figure 4.26: Behaviour inside of the `Coordinator` actor.

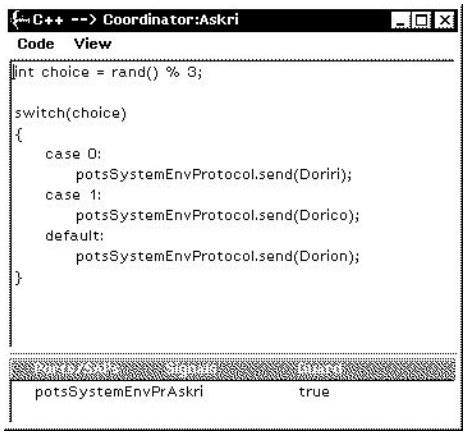

Figure 4.27: Co-ordination of the `Ringing` state of the `Environment` actor. The transition code for the `Askri` coordination request received by the `Coordinator` is shown.

and it is permissible under coordination rules.

For example, consider the `off_hook_dial` substate of the `Phone B` actor, as shown in figure 4.28. Six transitions in total are made into the `S1` state, including `OnOfA, OnOf, RiOf, SOS1, BuOf,` and `OnOfB`. It is therefore necessary to send the `AskOf` signal to the `Coordinator` actor in each of these transitions.

Non-local choice

If the first sender of all of the successors of a substate is the same process, then the branching decision is *local*. That process can autonomously make a decision as to what action to take, and all other processes proceed with the wait-and-see strategy, discussed earlier. Hence, there is no need for global synchronization of all processes at the branching point and the need for them all to understand the branching strategy. In this case, the global coordinator simply acts as a entity for randomizing the local branching choice.

However, the need for a global coordinator becomes clear in the presence

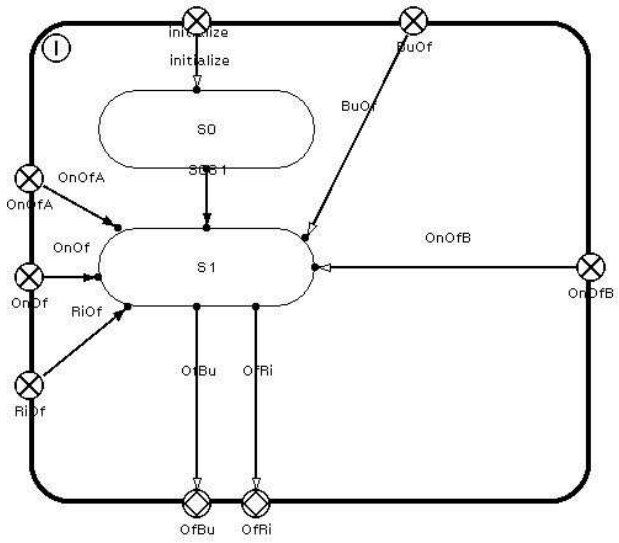

Figure 4.28: The `off_hook_dial` substate of the `Phone B` actor.

of non-local choice in an under-specified MSC specification [6]. For example, consider the removal of the extra synchronization message mentioned in footnote 5. This will cause the branching type to be changed from local to *non-local* (either `Phone A` or `Phone B` may initiate a disconnect — there is no unique first sender in the `on_hook_term` and `on_hook_orig` bMSC's). The under-specification here is that the model will not function if both phones initiate a disconnection sequence at the same time — if phone A disconnects by sending a request to B, then receives a disconnection request of its own from phone B immediately after, it will not reject it, as this synchronization is not specified. Only one or the other disconnect is intended to occur and be validated at a time, hence the need for coordinating which is initiated first.

Coordination becomes possible when each actor in the system encounters a branching decision where non-local choice is present, stops, and requests coordination by the global coordinator actor. This entails each participat-

[6] As explained earlier, it is expected that underspecified process synchronizations will be rectified during the design process.

ing actor sending an `Ask` signal to the coordinator at the branching point. The coordinator maintains an internal counter and increments it on reception of each coordination request. Once all actors have "reported in," the coordinator makes its choice and informs all actors of its decision by a `Do` signal broadcast. The mechanism relies on the fact that under the maximum traceability approach, each state in the hMSC specification is duplicated in the finite state machines of all actors, and hence it is possible to determine and match the current state of each actor and identify the global coordination point. In the case of local choice, only one actor requests a branching decision and coordination of other processes is unnecessary.

The automatic coordination being proposed is a simplification of the synchronizing history variable mechanism [40, 59], outlined in Section 4.5.3. It does not maintain a historical record of branching decisions made by each actor, thereby reducing its state space. Process divergence, whereby one actor can visit the same branching point several times ahead of the other actors, is not allowed, as the global coordinator requires that all actors arrive at the same branching point at the same time before making a decision. Process divergence can always be detected at the specification stage by using MESA [59] and eliminated before synthesis.

It is possible to implement finite-length synchronizing variables in ROOM, but they consist of data that must be globally accessible by every actor. A containing entity, such as the coordinator actor, must provide access. The coordinator can be programmed to handle all history variable updates and checks while retaining the `Ask` and `Do` message interface. This isolates the coordination logic from the application actors to reduce their complexity.

Synchronizing semantics should always be avoided. The beginning and end of a bMSC should not be interpreted as synchronization points for all processes. The automatic coordinator complies with this rule, as it simply replaces message injections of [5] at local and non-local branching points, which already avoid synchronizing semantics. Processes that can resolve their branching path through the wait-and-see strategy do not participate in

the coordination. Only those processes that cannot do so request instruction from the coordinator.

Generalization

The injection triggers created by the standard maximum traceability algorithm are replaced by coordination requests. An interface to the global coordinator actor is synthesized for every actor that sends a coordination request. In the case of non-local choice, each actor must send a coordination request to the coordinator once the branching point is reached, if that actor is the first sender in a successor bMSC (the other actors use the wait-and-see strategy: they react based on the type of message received). Once all requests from all actors have been received, the coordinator makes its choice and replies to all actors.

Algorithm

The following is an algorithm for generating a global coordinator actor, as a modification to the maximum traceability algorithm in [5]:

1. **Approach:**
2. The maximum traceability algorithm assigns a timeout signal for every transition corresponding to a message being sent. The timeout signal is then replaced by an injection point if it is the first transition in the sub-ROOMchart. Here, the injection signals will be replaced by coordination signals.
3. **Algorithm:**
4. Create the `Coordinator` actor on the system-level ROOMchart.
5. For each actor A_i in $A = \{A_0, \ldots, A_n\}$, the set of all actors in the system:
6. For each statechart $SubRC_i$ (corresponding to a bMSC) of A_i:
7. If the number of outgoing transitions of the last state s_l of chart $SubRC_i$ is > 1 (i.e. s_l is a branching point with at least two outgoing transitions), and s_l is identified as a local or non-local branching point:

8. Create an outgoing signal called $Ask\{Name_{curr}\}$ to the `Coordinator` actor, where $Name_{curr}$ is the unique name of the substate $SubRC_i$. Add a send action for this signal to the transition code of all transitions into state s_l.

9. For each outgoing transition O_t of $SubRC_i$ (from state s_l):

10. Enable the O_t superstate exit transition (to successor superstate $SubRC_s$) with the signal $Do\{Name_{succ}\}$ received from the `Coordinator`, where $Name_{succ}$ is the unique name of the successor superstate corresponding to the O_t transition.

11. Create a corresponding transition called $Ask\{Name_{curr}\}$, enabled by the $Ask\{Name_{curr}\}$ signal, in the `Coordinator`, if it does not already resist. Add the $Do\{Name_{curr}\}$ signal to the randomization code for this transition.

12. Create protocol P_{ic} and add all newly defined $Ask\{Name_{curr}\}$ and $Do\{Name_{succ}\}$ signals to the outgoing (for the `Ask`) and incoming (for the `Do`) signal lists of the protocol bound to the `Coordinator`. Create end ports and a binding between actor A_i and the `Coordinator` actor implementing protocol P_{ic}.

13. If the branching decision is non-local (and the $Ask\{Name_{curr}\}$ transition is already defined in the `Coordinator`), then increment an internal counter *cnt* in the transition code. When *cnt* equals the total number of coordination requests that are expected by all actors for that branching point, then the Coordinator is to proceed with the random choice of `Do` commands. Generate the appropriate `if/else if/else` construct for the counter check.

Example

Re-examine figure 4.24. The `coordinator` actor is first synthesized on the system-level structural chart. Now, starting with the `Environment` actor, all of its super-states, or statecharts, are traversed. Suppose that the current statechart being examined is `Ringing`. As shown in figure 4.25, the last substate of this statechart is S_2. Three transitions out of this state are defined, including `RiOn`, `RiCo`, and `RiRi`. The destination super-states are `on_hook_ringing`, `conversation`, and `ringing_timeout`, respectively, as can be seen from figure 4.21. Therefore, `S2` is identified as a branching point.

Next, a coordination request signal, called `Askri` (an abbreviation for "Ringing"), is created. The signal is sent to the `coordinator` actor in the transition from S_1 to S_2. Only one such incoming transition into S_2 is defined.

Next, for each of the three outgoing transitions from state S_2, the enabling condition is programmed as being the receipt of a `Do` signal from the coordinator. For instance, the `RiOn` transition is triggered by the receipt of a `Dorion` (an abbreviation for "Ringing to On-Hook-Ringing") command signal.

When the first outgoing transition, `RiOn`, is processed, a corresponding transition, called `Askri`, is synthesized in the `Coordinator` actor, as pictured in figure 4.26. In the code for the transition, shown in figure 4.27, one of three `Do` signals is randomly selected to be sent to the `Environment` in response to the `Askri` request. Each additional outgoing transition that is processed in the `Environment` actor, after the first, is added to the code block of the existing `Askri` transition in the `Coodinator`, i.e. the processing of `RiOn` causes the `Askri` transition in the `coordinator` to be synthesized, and the `RiCo` and `RiRi` transitions are added to the switch statement of the existing transition code, so that only one transition in total is created in the Coordinator for the `Ringing` superstate of `Environment`.

The three new Ask signals are added to the outgoing signal list of the `potsSystemEnvProtocol` (from the view of the `Environment` actor), while the Do signals are added to the incoming signal list of that protocol. Finally, end ports and a binding are synthesized between the `Environment` and the `Coordinator`.

Implementation

The global coordinator actor synthesis algorithm has been implemented in C++ for the Maximum Traceability synthesis approach in the ROOM synthesis engine of Mesa. The implementation includes the structural and behavioral synthesis of the coordinator actor, and its coordination activity

with other actors in the system. `Ask` and `Do` coordination signals and events are automatically synthesized, and the coordinator deterministically chooses the first out transition that is created for all local and non-local branching points, thus coordinating all actors in the system. The coordinator instantly issues the coordination command signals in response to requests from any actors participating in a branch. The execution of the model is now fully automated during simulation, and no longer requires user intervention through message injection. The implementation has been tested and verified, through simulation, on a model exhibiting both local and non-local choice.

Applications

It is possible to replace the random branching decisions of the coordinator with user-programmed scenarios, by having the coordinator read the decisions from an input simulation file. This is a useful technique for performing discrete event simulations, and for manual validation of scenarios.

Another effective use of this coordination mechanism is in performance analysis. If the probability of all branches occurring in the system is not equal, then a choice can be biased by specifying appropriate ranges in the checks against the random number being generated.

Multiple coordinators

Only one coordinator is synthesized in the system. If a coordinator instance was synthesized for every actor requiring coordination, then it would be possible for non-local choice to result in conflicts. For example, consider the `Conversation` state of all actors. If `Phone A` and `Phone B` were independently allowed to terminate a connection on their own, then a situation could arise whereby `Phone A` and `Phone B` would simultaneously proceed with their disconnection sequence, causing a loss of synchronization.

Also, it would be impossible for `Phone A`'s and `Phone B`'s coordinators to mutually agree on a single action to be taken, such as only `Phone A`

disconnecting, without introducing complicated arbitration logic.

Coordination is simplest in the absence of non-local choice. A non-local choice can be transformed into a local choice by allowing only one actor to decide a successive course of action, i.e. requiring a unique first sender in all successors to a branching node — this is always possible in a completely specified deterministic system with the environment modeled.

For instance, as specified in the MSC model, the environment alone decides whether `Phone A` or `Phone B` disconnects, by sending an `OnHookTerm` signal to `Phone B`, or an `OnHookOrig` signal to `Phone A`, respectively, as opposed to the unrealistic scenario of having either phone handler autonomously decide its own course of action.

Tool support

Standard C libraries must be added to the project in the configuration browser window, in order for the randomization function to be compiled correctly. Otherwise, the entire structure and behaviour of the coordinator can be automatically generated within a linear form file.

Other work

An interface for an actor control mechanism has been proposed, but only in the context of system start-up and reset for the purpose of system activation, failure recovery, and preventive maintenance [12]. The co-ordination of activities within the system based on an intimate knowledge of the global finite state machine has not been addressed.

4.6 Simulation of ROOM models

A useful application of the global coordinator is in the simulation of a ROOM model. It is possible to quickly generate execution traces that can be used

for validation of properties, as the coordinator automatically synchronizes all actors, and eliminates the need for user intervention as in [5]. The traces can then be used for validation of properties, on an implementation-level. This section provides insight into the process.

OTD allows the software designer to generate a design-time MSC by using the internal message tracing facility during a simulation run. The MSC is not created automatically, but it can be viewed by configuring which actors to monitor, then running a run-time system simulation to create a trace. A sample MSC is shown in figure 4.29. In this scenario, `Phone A` calls `Phone B`, which starts ringing, but `Phone A` hangs up before `Phone B` is picked up. A useful procedure in testing is to compare the MSC's generated from a design to those produced from the initial requirements stage. This illustrates the fact that an important use of MSC's is in the creation of test plans and design validation.

For example, a design-time and a run-time MSC can be compared in order to validate an implementation. OTD offers a facility called the "Differences" tool to report any variation that it finds. The trace can be exported, but the format does not match that of Mesa.

An unfortunate limitation of OTD is that states may also be shown on the design-time MSC, but only the leaves, as in figure 4.30. It is therefore difficult to trace the states drawn to their higher-level counterparts, as the state S_1 is found in every hierarchical state, for example.

4.7 Conclusion

In Section 4.1.3, a number of criteria were identified with which to evaluate the current ROOM synthesis algorithms [5], and propose improvements so that they may be satisfied to a greater degree. Hand-coded ROOM models were created from scratch following the algorithms proposed in this chapter, and illustrated throughout the text. The following is an analysis of how the proposed features addressed the various criteria:

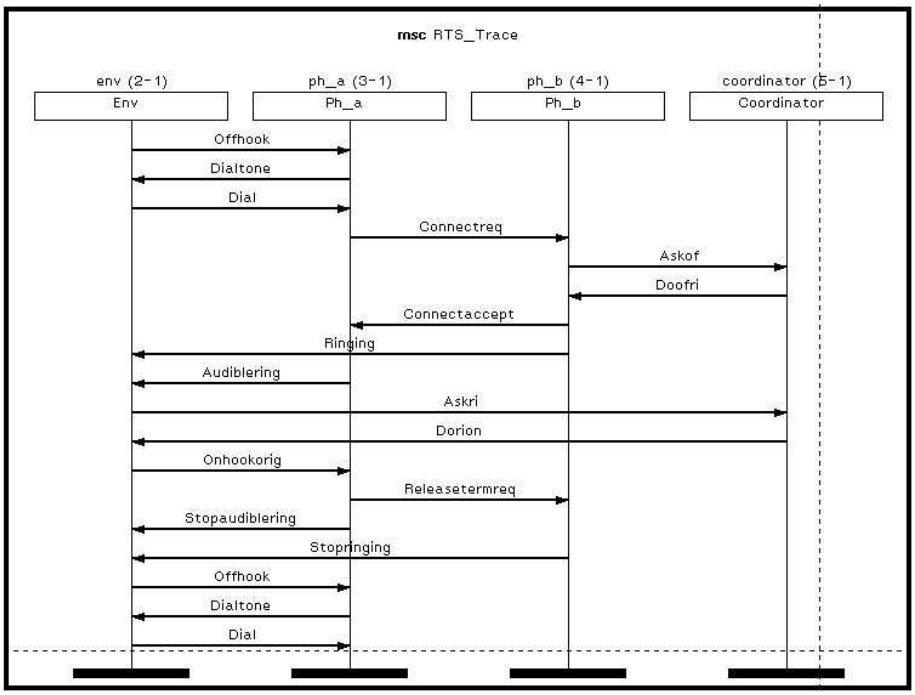

Figure 4.29: Design-time MSC generated from a simulation run, involving automatic coordination (see Section 4.5.5). Here, `Phone A` dials `Phone B`, and requests a coordination command from the `Coordinator` by sending the signal `Askof`, following the example in 4.5.5. The `Coordinator`'s response is `Doofri`, instructing the system to branch to the Ringing state (corresponding to the `Ringing` bMSC), and the `ConnectAccept` message is sent to `Phone A` as the first message in this state, accepting the incoming call request. A little later, coordination occurs once again when the `Environment` sends the `Askri` signal, requesting a branching decision from the `Ringing` state. The response, in the form of the `Dorion` message, signifies that the originator places the phone back on-hook, and control proceeds to the `On_Hook_Ringing` state. The rest of the call activity is not shown in the MSC.

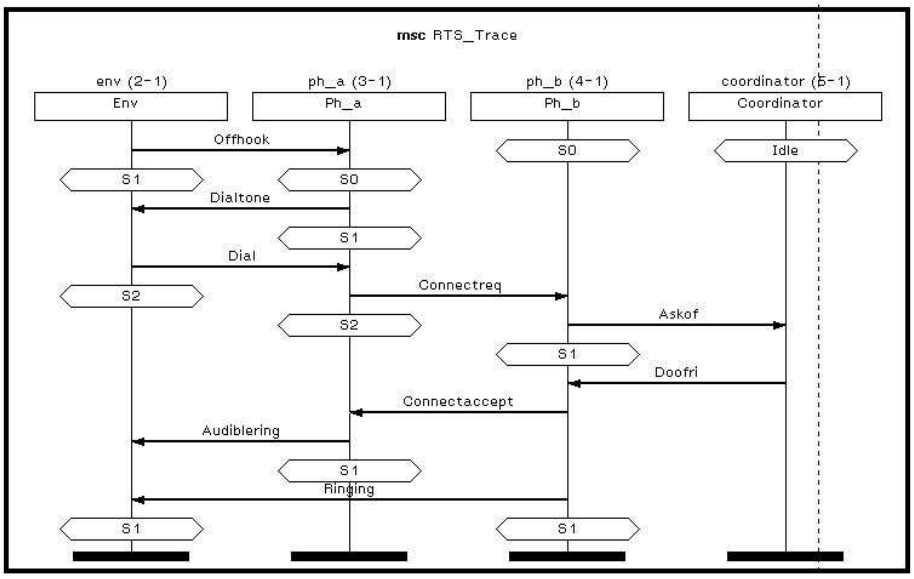

Figure 4.30: Design-time MSC with substates displayed. The same example as in figure 4.6 is shown, but with additional top-level state information displayed.

Traceability. The mapping of bMSC states to actor states was investigated in Section 4.5.1, and no improvement in the locality of transitions of the maximum traceability algorithm was found to be possible.

Understandability. Multiple containment, in Section 4.2.6, allows different interfaces of the same actor component to be represented in different abstractions, so that parts of its functionality can be isolated for viewing, increasing comprehension.

Maintainability. The grouping of actors into hierarchical structures, discussed in Section 4.4.1, can assist maintenance work by isolating views at different levels to support unit testing of sub-actors, and also helping to localize errors. The greater abstraction that is possible also improves understandability. The automatic coordinator of Section 4.5.5 eliminates the need for user-injected control messages for branching decisions, and supports the execution of preprogrammed test scenarios, thus simplifying and quickening the process of testing.

Complexity. In the discussion of non-determinism in Section 4.5, it was not found to be advantageous to replace message triggers for multiple state exits with choice points, as the number of sub-states in and subsequent complexity of each actor state would be increased. Multiple containment and hierarchical actor structure, however, permit an actor and its interfaces to be viewed at different levels of abstraction, reducing the apparent complexity of the view. No significant difference in complexity was found to exist between the maximum progress and maximum traceability algorithms [5].

Correctness. The global coordinator mechanism automates branching decisions during simulation of a model, allowing a trace log to be quickly produced to validate its behavior. Syntactic notation for MSC's has been suggested for hierarchical actors and replication factors, and all of the synthesis algorithms can be implemented so that the synthesis process is fully automated using a tool such as Mesa, thus reducing the potential for human errors in the translation.

Extensibility. The merging of finite state machines of different process instances, i. e. views, described in Section 4.2, supports the creation of different structural architectures (e.g. call-based versus phone-based) from existing process descriptions. Replication, in Section 4.3, allows more complex scenarios to be run by instantiating the same actor class more than once. The algorithms have been designed to be compatible with each other, so that it is possible to create replicated, structurally hierarchical actors, for instance.

The hierarchical actor structure, replication, finite-state-machine merging, and global coordinator synthesis algorithms were implemented in C++ and Tcl/Tk and verified in the Mesa tool, proving the validity of the algorithms.

Chapter 5

Validation of Requirements for GSM Mobility Management using MSC's and Temporal Logic

5.1 Introduction

5.1.1 Background

As explained in Chapter 1, software projects in the area of real-time telecommunication systems are highly complex, and the accurate specification of requirements on the externally observable behavior of these systems is an important step in their development. Studies have found that most of the software development failures that occur are due to errors introduced in the software requirements and specification phase [47]. Therefore, it is important to demonstrate a methodology of proving that a system design consistent with initial specifications fully satisfies all desired functional properties, before proceeding to the implementation stage.

The system under study in is an existing European cellular telecom-

munications system called GSM (often referred to as the Global System for Mobile Communication), composed of several entities whose functions, properties and interfaces have been well-defined and standardized by ETSI (the European Telecommunications Standards Institute) [48], [49], [54], [57]. The focus is on formally capturing and validating requirements on the communication protocol aspect of the interaction between these entities with respect to a service called *Mobility Management*. The formal specification and validation of the properties of this system is important to detect conflicting and incomplete requirements specifications, and to provide unambiguous documentation and useful groundwork for future transformation of the system requirements to a design and implementation in the later stages of the life cycle.

5.1.2 Approach

As a first step, Message Sequence Charts [55, 56] were used to capture the majority of the operational requirements of Mobility Management of GSM. MSC's were chosen due to their simplicity in the specification of message-passing concurrent systems, and their prevalence in the field. A thorough discussion of the MSC language is found in Section 2.1 on page 24. MSC's were only used to specify the operational behavior of the system, not any specific properties that were required to be held. A number of safety and liveness properties were identified as being required of the system, but could not be validated directly through the MSC's alone. Therefore, the specification required translation to another type of model that could be used to (through a tool) automatically validate the properties. An operational model was built from the MSC specification using the Promela language and was simulated and validated using the Spin tool. The informal high-level requirements from the system requirements were encoded in LTL (Linear Temporal Logic) formulæ [53]. These were used to determine whether all required properties hold of the Promela operational requirements model through the use of partial and exhaustive state space exploration techniques of the Spin model checker [50].

Although the initial prototype was built from scratch by hand and validated as a proof-of-concept, a later version was fully synthesized through the use of automated tool support, arriving at an equivalent model (in terms of process type definitions and message exchanges), which also validated correctly. The use of CASE (Computer-Aided Software Engineering) tools at all stages of the process described herein greatly validates the approach as a viable and practical methodology in the field.

5.1.3 Related work

π calculus and Lotos

A formal specification and validation of the mobility management service of GSM has been previously presented using π calculus [60] and LOTOS [63, 64]. There is also an automatic tool available named Mobility Workbench, which is based on π calculus. The Mobility Workbench allows for the analysis of mobile concurrent systems [62]. The approach prescribed in this thesis is based on MSC's as a front-end to the specification of the operational behavior of mobility management. This approach makes it simpler to capture the essentials of message-based protocols compared to beginning with a low-level mathematical specification method such as π calculus or LOTOS. The visual MSC specification is then translated into Promela [50] and analyzed using the SPIN tool [51] for compliance with a set of LTL-formalized high-level requirements. While the verification tool based on the π calculus has the ability to detect deadlocks, it does not allow for the validation of arbitrary safety and liveness properties as in the approach described here.

5.2 Overview of GSM

A simplified architecture of the GSM network is shown in figure 5.1. A description of each subsystem follows in Section 5.2.1.

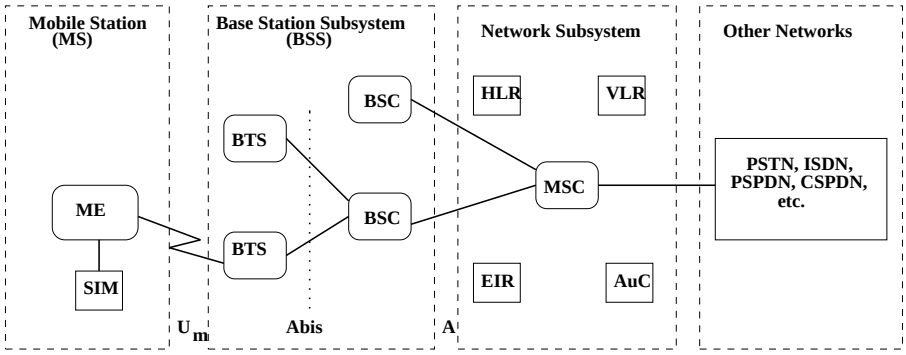

Figure 5.1: Architecture of the GSM Network [58].

5.2.1 Structural description

Cellular telecommunications is one of the fastest growing telecommunications applications of the present age, and has already become a prominent way of personal communication. GSM is a digital cellular telecommunications system, a standard created by ETSI [48, 49]. GSM provides standardization to guarantee proper inter-working between components of the cellular system, achieved by providing functional and interface descriptions for each of the functional entities defined in the system. The *Bearer Services* of GSM dictate the means by which the user communicates with the network, and *Teleservices* define the communication between users. The basic teleservice provided by GSM is *telephony*. GSM users can also send and receive data to and from POTS, ISDN, PSPDN and CSPDN networks using a variety of protocols [57].

The general architecture of GSM is illustrated in figure 5.1. The MS (Mobile Station) in the GSM architecture is essentially a portable phone communicating with the BSS (Base Station Subsystem) through the radio frequency interface U_m. The BSS consists of a number of components, including the BSC (Base Station Controller), providing control functions, and the BTS (Base Transmitter Station), containing the radio transmitter. The BTS is used to provide radio coverage in a geographical zone called a cell, while the BSC controls a larger geographical area called an LA (Location

Area), consisting of multiple cells. This is made possible through the BSC controlling multiple BTS's. The BTS and BSC are connected through a standardized interface called *Abis*. The BSS in turn communicates with the MSC (Mobile-services Switching Center), containing switching functions, through the interface *A*. The MSC handles the switching of calls between mobile users, and between mobile and fixed network users. In addition, it manages the location registration procedure for wandering mobiles. The MSC stores two databases: the HLR (Home Location Register), containing all subscriber information, and the VLR (Visitor Location Register), dynamically storing only the subscriber information for mobiles located in the area being serviced by it. The MSC includes two other registers: the EIR (Equipment Identity Register) and the AuC (Authentication Centre). The EIR contains a list of the IMEI (International Mobile Equipment Identity) numbers of all valid mobile stations on the network. The AuC stores a copy of the secret key used for authentication that is stored in a user's SIM (Subscriber's Identity Module). Overall, the general subsystem under study in this thesis is the SMSS (Switching and Management Subsystem), responsible for signaling protocols by which calls are established, maintained, and cleared.

5.2.2 Mobility management

The Mobility Management (MM) layer is involved in keeping track of the mobile while on the move, and handles the operations which arise due to the movement of the subscriber across location areas, as well as his interaction with the network at all times. It is also concerned with the location update procedures that enable the system to realize the current location of a switched-on MS so that incoming call routing can be performed successfully. Authentication and security issues are also handled, as are call management procedures.

There are two possible states of a mobile station: turned off or turned on. If the MS is turned on, it may be either in the idle or in the conversation

state. The following is an overview of the behavior of the system with respect to these states:

- *MS is turned off:* The MS is simply inoperative and does not update the system about its location. It cannot be reached by the network.
- *MS is turned on but in the idle state:* The MS is considered to be attached to the system, and it can be reached through paging by the network. While roaming, the MS must always be connected to the network, and to ensure this, it must update its location in the system should it move to a new cell. The mobile is considered to always be switched on, synchronized to the system, and ready to send or receive a call.
- *MS is in conversation state:* The MS has already been allocated traffic channels i.e. it is connected to the network. The handover procedure will allocate a new traffic channel if the signal on the current channel drops below an unacceptable level.

One of the major characteristics of mobility management is that the mobile station may be connected to different radio channels in the system at different points of time. This feature is called handover. Due to its importance in mobility management, this aspect will be discussed in the following subsection. The other functions related to mobility management such as security, location update, connection establishment and disconnection can easily be understood from the corresponding bMSC's in figure 5.6 on page 152 and signal dictionaries in Section 5.4 on page 131.

5.2.3 Handover

Call handover is the process by which a call in progress is automatically transferred to a new cell. It can occur when the loss of the call is imminent due to movement of the mobile and subsequent drop in signal strength. The three purposes of handover are: alleviation of excessive interference

within a cell, that of between cells, and the easing of traffic congestion by moving calls between cells. Thus, handover can occur between channels of the same BSC, between channels of different BSC's of the same MSC, and between channels under the control of different MSC's in the same PLMN. The BSS alone may handle the connections for handover within the same cell or between the cells under its own coverage. This is called *internal handover.* In the case of handover between cells under the coverage of two different Base Stations, the MSC gets involved, and this kind of handover is called *external handover.*

In this paper, handover is considered between different base station controllers under the control of the same MSC. This is called intra-MSC handover [48]. In external handover, the MSC uses the signal quality information reported by the MS and preprocessed at the BSS. The original MSC will always keep control of the call in an external handover to a different and even a subsequent MSC [54]. The handover mechanism implemented in the model works as follows [48]:

> If the BSS to which the MS is currently connected determines that the MS is required to be handed over to a new BSS, it will send a *handover required* message to the MSC. This message contains an ordered list of preferred cells based on some predefined handover criteria. Upon receiving the above message, the MSC initiates the handover process by sending a *handover request* message to a new BSS, which will in turn acknowledge the request by a *handover request acknowledge* message to the MSC. The MSC will then inform the MS of the channel belonging to the new Base Station by a *handover command* message to the MS via the present BSS. After receiving the *handover command* message, the MS disconnects the channel of the present BSS and initiates connection with the channel of the new Base Station by sending a *handover access* message to the new BSS. If the MS is successfully connected to the new Base Station, the new BSS will send a *handover detect* message to the MSC. After that, the

MS will also send a *handover complete* message to the MSC via the new Base Station. Upon receipt of the *handover complete* message, the MSC sends a *clear command* message to the old BSS, which can allocate the freed channels to another MS. The old BSS will respond to this message by a *clear complete* message. If the MS is unable to connect to the new Base Station, it will try to re-establish the connection on the old Base Station. If this succeeds, the MS sends a *handover failure* to the network via the old Base Station, and resumes operation as before the handover attempt. If the MS also does not succeed with the old Base Station, the MS is disconnected from the network. For simplification, in the model presented here, the ideal case is considered, where the MS will always be able to connect with the new Base Station. The message sequences used in the model are shown in the `Handover` bMSC (basic Message Sequence Chart) of figure 5.6 on page 152.

5.3 Validation tools

5.3.1 Promela/Spin

Promela (PROcess MEta LAnguage) [50] is a design-based modeling language that allows the specification of the behavior of asynchronously executing concurrent processes that may interact through synchronous or asynchronous message-passing or through direct access to shared variables. A Promela model can consist of three different types of objects: processes, variables and message channels. Promela was developed at Bell Laboratories, and is used as an input language to the validation tool Spin.

Spin (and its graphical version XSpin) [50, 51] is a widely-used software tool supporting the formal validation of distributed system software. It has been used to trace logical design errors in distributed and concurrent system designs, including telecommunication systems. It is used to check the

logical consistency of the specification and report any deadlock, unspecified receptions, incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

5.4 Construction of model

A formal Promela Model has been created to capture a simplified functional model of the Mobility Management features of GSM using Message Sequence Charts. The choice of MSC's is particularly appropriate given the nature of Mobility Management in that it mainly concerns a message-passing protocol.

5.4.1 Architecture of the model

The high-level architecture of the Promela model is based on the simplified GSM architecture presented in figure 5.1 on page 126. The current GSM specification standard has been inspected, and a reasonable level of abstraction has been chosen for the work in mapping GSM components into entities, to preserve as much accuracy as possible in the model without unnecessary complexity. The entities of the model include the `MS`, `BSS`, `MSC`, and `Network`. The Promela model is composed of the following processes corresponding to these entities, previously described in more detail in Section 5.2.1 on page 126:

- `MS.` The mobile station, e.g. a wireless phone.
- `BSS.` The base station subsystem, communicating with a mobile via radio frequency.
- `MSC.` The mobile services switching center.
- `Network.` The network subsystem. This process encapsulates the activity generated by users of the network, e.g. an ISDN modem user initiating calls to the mobile.

Each process represents the control software running on each respective hardware subsystem. For example, the BSS represents the control software running on the base station transceiver assembly. The interface between the processes consists of messages, and channels of sufficient capacity (bounded channels).

Basic MSC	Description
Chan_Req	The mobile sets up a communications channel to the base station. It is then considered to be synchronized and connected to the network.
Call_Set_Req	The mobile creates a service request for initiating a call to another party.
Page_Res_Req	The mobile creates a service request by responding to a page from the network.
Loc_up_Req	The mobile creates a service request to update its location in the network based on the signal strength of all nearby base stations.
Identity	The network confirms the identity of the mobile, and retrieves the subscriber information for a newly arrived mobile.
Auth_Req	The network authenticates the user through a security key challenge.
Auth_Succ	The user's request is confirmed and access rights have been verified.
Auth_Fail	The user's request for the service request has been denied.
Mob_Init_Call	The mobile initiates a call to another party.
Mob_Pag_Res	The mobile answers an incoming page as part of an incoming call.
Loc_Up	The mobile's location in the network is updated.
Mob_Init_Disc	The mobile terminates the call.
Net_Init_Disc	The network terminates the call.
Handover	The MS connects to a new BSS due to its location change during conversation.

Table 5.1: Description of the bMSC's making up the GSM mobility management system behavior.

The following is a dictionary of all of the signals in the system, categorized by the bMSC in which they appear:

Chan_Req
The mobile is turned on and connects to the network.

Message	Source	Destination	Description
chan_req	MS	BSS	Request for a communications channel from the base station.
chan_assign	BSS	MS	A channel is granted and synchronization is made.

Call_Set_Req
The mobile initiates a call to another party.

Message	Source	Destination	Description
cal_chan_req	MS	BSS	Request for service for initiating a call by dialing the other party's PSTN number.
cal_chan_resp	BSS	MS	A call channel is granted.
cal_set_req	MS	MSC	A request to initiate a confirmed.
cal_set_req1	MSC	Network	If the called party is a PSTN/ISDN user, his device begins to ring. Otherwise, if the called party is a mobile, a page request will be initiated.

Pag_Res_Req

The mobile responds to a page, preceding an incoming call.

Message	Source	Destination	Description
pag_chan_req	MS	BSS	A mobile will monitor all incoming page requests and answer only those specifically addressed to it. A request for a channel assignment is sent in response.
pag_chan_ack	BSS	MS	The call channel has been granted.
pag_res_req	BSS	MSC	A paging response is forwarded to the network over the assigned channel, and through the switching centre.
pag_resp	MSC	Network	Completion of the paging response.

Loc_Up_Req

The mobile, while idle, detects a loss in signal strength. It selects a stronger base station, and updates the network of its corresponding location.

Message	Source	Destination	Description
loc_up_req	MS	BSS	The mobile monitors the received signal strengths of all base stations. Should one drop below an unacceptable level, a location update request is made.
loc_up_req1	BSS	MSC	The location of the mobile is updated in the switching center's database.
loc_up_req2	MSC	Network	The network is notified as well.

Identity
The mobile moves to a new location area, and must be identified by the network.

Message	Source	Destination	Description
iden_req	Network	MS	The network does not recognize the identity of a mobile that has just arrived in its location area. A request is made for the mobile's IMSI.
iden_res	MS	Network	The IMSI is provided.

Auth_Req
The mobile has requested service, and it must first be authorized.

Message	Source	Destination	Description
auth_req	Network	MS	An authentication challenge request, containing a challenge word.
auth_resp	MS	MSC	The mobile computes a challenge response based on its key.

Auth_Succ
The mobile has been identified and its access rights have been verified.

Message	Source	Destination	Description
serv_accept	MSC	MS	The challenge response is verified, and the request for service is accepted.
serv_accept	MSC	Network	The network is notified.

Auth_Fail
The authorization has failed.

Message	Source	Destination	Description
auth_fail	MSC	MS	The challenge response is found to be incorrect. The request for service is denied.
auth_fail1	MSC	Network	The network is informed.
rel_chan	MS	BSS	The channel is released, only in the case of a call or page channel request (which previously resulted in a call channel allocation).

Mob_Init_Call

The mobile, once authorized to originate a call, initiates the connection procedure.

Message	Source	Destination	Description
assign_cmd	MSC	MS	Indication that the call can proceed, and an assignment of the call channel.
assign_compl	MS	MSC	The assignment has completed.
init_addr_mob	MSC	Network	The network is updated of the call service request and the number being dialed.
adr_compl	Network	MSC	The called party has been alerted to the incoming call.
alert	MSC	MS	The mobile is also informed of the called party being alerted to the incoming call.
answer	Network	MSC	The called party has answered the call.
connect	MSC	MS	The mobile is informed of the other party answering the call.
conn_ack	MS	MSC	The mobile acknowledges the connection, and conversation begins.

Mob_Pag_Res

The mobile answers an incoming call.

Message	Source	Destination	Description
cal_setup	MSC	MS	The calling and called parties' numbers.
cal_conf	MS	MSC	Confirmation.
alerting	MSC	MS	The mobile should begin to ring.
alert	MS	MSC	Confirmation that the mobile has begun to ring.
connect	MS	MSC	The call has been answered.
con_ack	MSC	MS	Verification of the call answer notification.
answer	MSC	Network	The mobile has accepted the call.

Loc_Up

The mobile has moved to a new location area, and the network must be updated of its new location.

Message	Source	Destination	Description
update_loc	MSC	Network	The network is informed of the previous location area of the mobile.
loc_upd_result	Network	MSC	The result of the location update.
loc_upd_result	MSC	MS	The result of the location update is passed on to the mobile. Either the mobile has been identified and its roaming rights have been verified, or its identity or location cannot be confirmed, and service is rejected.

Mob_Init_Disc

The mobile terminates a current call.

Message	Source	Destination	Description
discon_req	MS	MSC	The mobile disconnects from the call.
release	MSC	Network	The release is forwarded to the network.
rel_conf	Network	MSC	Network verifies receipt of the release request.
rel_comp	MSC	MS	The call channel is ready to be released.
rel_chan	MS	BSS	The previously allocated call channel is now released.

Net_Init_Disc *A network user, engaged in a call to a mobile, terminates the call.*			
Message	Source	Destination	Description
release_req	Network	MSC	The network user has disconnected from the call.
release_req	MSC	MS	Forwarded disconnect request to mobile.
release_chan	MS	MSC	The call channel is ready to be released.
connect	MSC	Network	Notification that the connection has completed.
rel_conf	MSC	Network	Notification that the call channel will be released.
rel_comp	MSC	MS	A release complete acknowledgment.
rel_chan	MS	BSS	The call channel is released.

Table 5.2: Signal dictionary of the GSM system.

5.4.2 Specification of requirements

The behavior of the GSM Mobility Management system is described using MSC's. A hierarchical hMSC is presented in figure 5.4 on page 150 in order to depict the top-level abstraction of the GSM Promela Model. The hMSC's in this figure are further elaborated in order to show the control flow between all of the bMSC's. These hMSC's are as follows:

- `Loc_up_Req:` The network is updated with the new location of the mobile.
- `Service_req:` Channel allocation and request for service.
- `Authorization_loc:` Identification and authorization for location update service request.
- `Authorization:` Identification and authorization for call setup and page response service requests.

- `Loc_update:` Location update.
- `Conversation:` Call setup and handover between two parties.
- `Termination:` Call termination.

The hierarchical approach is useful for describing large and complex systems in a modular fashion. Each bMSC is related to an operational function of the GSM specification for Mobility Management. Next, bMSC's are presented in figure 5.6 on page 152. Also, a description of each bMSC is provided in table 5.1 on page 133. The operational Promela model was constructed according to the behavior specified in the bMSC's. All of these diagrams were drawn using the MESA tool (see Section 1.6.1 on page 12).

5.4.3 Message simplification

The mobility aspect of the system was the central focus of this model, and as a result, a number of simplifications and abstractions were made with respect to the messages that were recorded in the MSC specification. The result is a compact representation of the actual, complicated GSM protocols described in the original specification. The simplifications included the following:

- The merging of two or more messages into a single message and the replacement of the same messages forwarded across one or more entities by a single message originating at the source and being received at the final destination. The application of these heuristics is demonstrated in figure 5.2.
- Another simplification made was the elimination of messages used for low-level hardware purposes, such as the synchronization of channels, handshake negotiation, framing, and other messages not central to the high-level concepts of mobility and call management. It was deemed entirely possible to model the general working of these features without complicating the protocols with many hardware-related messages. An example is shown in figure 5.3.

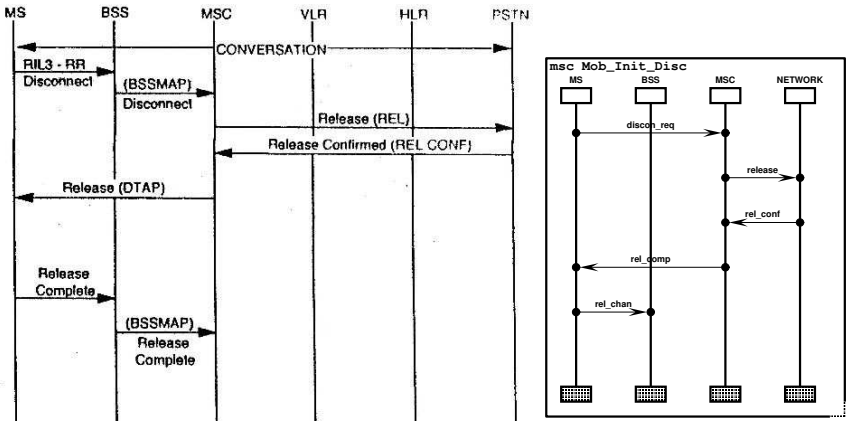

Figure 5.2: An example of a message-merging simplification on a mobile-originated disconnect is shown. The original MSC found in the specification [2] appears on the left, while the simplified version used for purposes of the Promela model appears on the right.

In the original specification, a `Disconnect` message is sent from the MS to the BSS, and then the same message is forwarded by the BSS to the MSC. This sequence was simplified in the project's corresponding bMSC. The two `Disconnect` messages were replaced by a single `discon_req` message. Similarly, in other MSC's, if the same message was forwarded across one or more entities, a simplification was made so that they were replaced by a single message originating at the source and being received at the destination. This pattern occurred frequently in the specification, and the simplification allowed for the reduction of many redundant messages to a more manageable number.

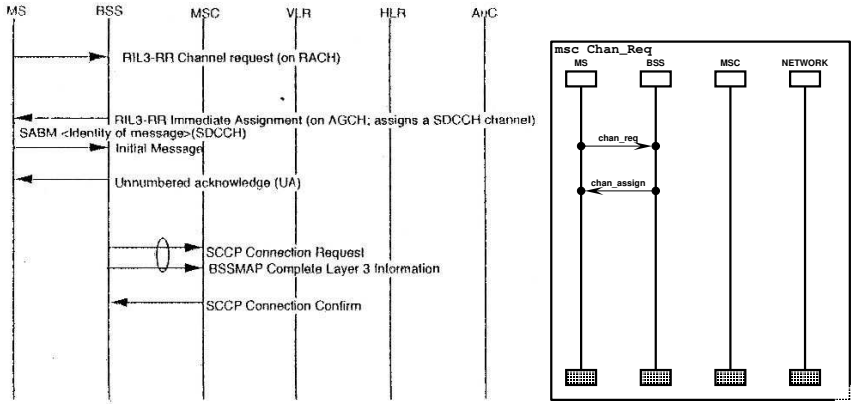

Figure 5.3: An example of a hardware message simplification on the `Chan_Req` bMSC is shown. The original specification [2] is shown on the left, and the simplified version on the right, as before.
This example illustrates that the request for a communications channel between the MS and the BSS involves a number of messages related to time synchronization bursts, frequency correction, framing, decoding of beacon frequencies, connection service requests on auxiliary access channels, and other particulars beyond the scope of high-level Mobility Management. The underlying meaning of these messages is a channel request between an MS and the BSS, and a subsequent confirmation. These ideas are encapsulated by the `chan_req` and `chan_assign` messages in the simplified `Chan_Req` bMSC. The elimination of the original hardware-related messages results in a more understandable model with a reduced state space that the SPIN tool will handle without great difficulty.

Simplification guidelines

The following message simplification guideline has been identified, and may be applicable to most systems:

1. If a message m is sent from process a to b, then the same message is sent from b to c (where a, b, and c are unique processes), then use transitivity to replace the two sends by message m sent from a to c. Process b must not have a message n ($n \neq m$) to send in response to receiving m, other than forwarding m to c, and the message m should not be used to resolve a wait-and-see strategy. Otherwise, removing the sends will stop the model from working. This is only a guideline, and other effects of removing the sends may be present.

While the simplification will result in a model being synthesized with fewer states for greater coverage in validation runs, the removal of messages may have an adverse effect on the functionality of the system.

5.4.4 Synthesis of Promela model

Once the MSC requirements specification was entered into the Mesa tool, its automatic Promela synthesis feature was invoked, resulting in a syntactically correct and complete Promela code file being generated. The following sections describe a number of important aspects of the synthesis process:

Communication

The MSC's were translated into Promela code following the approach described in [65]: every instance in the MSC specification was mapped to exactly one instance of a unique Promela process type definition. Each process was assigned a buffered channel, with a default capacity of one message, for its incoming traffic. When sending a message to a process, all senders used only the single channel specifically assigned to that process. The allocation of a single channel to a process is a rule of the synthesis, rather

than a property enforced by SPIN. The buffering of the channel allows for asynchronous communication.

MESA's synthesis feature supports a number of different channel definition schemes:

SDL: Each process has a unique input message queue assigned to it. This mirrors the characteristics of the communication model in the SDL language (for a detailed description of SDL, see Section 3.1 on page 35).

Point-to-point: A unique channel is assigned between every pair of processes for communication in each direction (and each channel can only be ever used by one sender process).

One-channel-per-message: Every message transmission is carried out on a unique channel, regardless of the sender or destination processes involved. It is also possible to individually assign messages to channels.

The SDL template results in the fewest channels being synthesized, and so is the best approach for keeping the resultant total state space to a minimum. The SDL template was the one that was chosen for the synthesis of the GSM system.

Static analysis

The synthesis feature performs a series of static syntactic analysis tests on the MSC specification before executing the translation into Promela. This includes verifying the connectivity of all bMSC's, identifying branching points where non-local choice is present, and finding and displaying any process divergence.

Synchronizing history variables

History variables (see Section 4.5.3 on page 104) were configured for all non-local branching points, with a default length of one, i.e. up to a maximum of one branching decision could be saved. The non-local choice points were

identified by running the non-local choice analysis feature of MESA. The MSC model was drawn so that all branching occurred out of connect points, rather than the start symbol, as history variables can only be defined for connect points.

An option is available in MESA to force the definition of history variables for all branching points in the specification to remain on the safe side, rather than adding them individually to non-local choice branching points. However, the addition of superfluous history variables increases the length of the state vector used in validation runs, and hence the potential for insufficient memory errors. As opposed to the five history variables defined manually, a total of eleven variables were created with the force option enabled — a significant inefficiency (as controlled by the user).

Process divergence was not found to be present in the model as presented in figure 5.6. Process divergence can cause a problem in a non-local choice situation, where a history variable needs to be kept and may be filled up as a result of the divergence. However, one difficulty that occurs is that the `Network` process takes no part in the `Handover` bMSC. Thus, it may try to initiate a disconnect by sending the `release_req` message to the `MSC` in the `Net_Init_Disc` bMSC, while a handover operation is occurring. The addition of a `hand_compl2` signal at the end of the `Handover` bMSC, sent from the `MSC` to the `Network` process, will notify the `Network` once the handover operation completes. The addition of this message adds a state to the `Network` process for the `Handover` bMSC so that a history variable can be added to it, resolving the problem. However, this has the side effect of introducing process divergence, whereby multiple consecutive handovers can flood the `Network` with the `hand_compl2` signal.

Instantiation

A difficulty that was encountered was that the Promela synthesis algorithms have no notion of the instantiation of process types, as previously mentioned. In modeling the key component of handover, this issue posed a problem. The handover being modeled involved the transfer of a mobile connection from one BSS to another, both being theoretically identical in operation but physically separated by distance. To model the interaction between

two independent BSS's, two different MSC processes were defined, `BSS` and `BSS_New`, although handover can in reality occur across any pair of base station subsystems in operation. In the precursory hand-coded model, the switch was performed by the MS and MSC processes keeping track of which BSS they were currently connected to for purposes of communicating with it by sending to either `BSS`'s or `BSS_New`'s channel. This was accomplished by setting a proxy variable to point to the correct channel, and sending all messages through it. The proxy was re-assigned during the handover operation, and so the mobile could switch between the two BSS's at will.

This special bit of logic was not present in the synthesized code. The `BSS` was instantiated as a process type supporting call negotiation (including channel allocation and release), while `BSS_New` was instantiated as a process type supporting only handover commands. The two behaviours could not be integrated into a single, coherent process type definition. Nonetheless, the working assumption was made that the `BSS_New` process would assume the identity of the new `BSS` process after the completion of the handover. In addition, the `BSS_New` instance was required to be defined as a silent process in all bMSC's other than `Handover`, as the synthesis requires that the same process definitions be present in all bMSC's in the specification.

Unordered receipt

A characteristic of the standard message receive mechanism in Promela is that messages are normally retrieved from the head of the input queue only. Therefore, all messages destined for a process must arrive in an expected order so that the process does not block. This initially caused a difficulty in the case of the `BSS`, which was attempting to initiate a handover operation by contacting the `MSC`, by sending the `hand_reqd` signal in the `Handover` bMSC, while the latter was still processing the mobile's authorization, in `Auth_Req`, `Auth_Succ` and `Auth_Fail`. Therefore, the handover request `hand_reqd` message waiting at the head of the `MSC`'s queue prevented the reception of the messages related to authorization (such as `auth_resp` from the `MS`), even if the channel buffer size was increased to hold all incoming messages. One solution that was tried was to add a synchronization message so that the `BSS` would be informed of the beginning of the conversation state. Another

solution was to use the *random receive* mechanism in Promela, in which the specific message expected in a state is retrieved from any location within the input message buffer. A third possibility would be to discard any unexpected message, but this would require a more complicated retry mechanism on the part of the sender.

Optimization

Because the length of the state vector ultimately affects the amount of physical memory used up during validation of the LTL properties, an effort was always made to reduce it whenever possible. The aim was to be able to perform validation of a model of comparable complexity to the GSM Mobility Management system on a typical current machine of 300-1000 MHz with 256 MB RAM.

Message send and receive events in the MSC's are modeled by the corresponding Promela statements with a special care to ensure the event atomicity by using the `atomic` scope of Promela, in which all processing runs to completion without interruption, unless message blocking occurs. Branching and iterations in the MSC specification are modeled through labels and `goto` statements in Promela. Non-local choice has been described in Section 4.5.2 on page 103.

An optimization that was made involved the merge of `atomic` blocks. The synthesis code encapsulates each send and receive event within an atomic block, making it impossible for the instructions within to be interleaved with those of another process, thus further optimizing the state space. This rule applies both to simulation and validation runs. However, it was observed that consecutive sequences of message sends and receives could instead be enclosed as a group within a single atomic block, as a further optimization. A filter was designed to process the code file and perform this tweak. However, an even more optimized version of the atomic block called a `d_step` could not be used in the model, as `goto`'s are not allowed into such a block. It was found that almost every state was accessed by a `goto` elsewhere in the process body due to the inherent branching in the generated code.

Example of translation

A portion of the translation of the `Chan_Req` bMSC of figure 5.6 to Promela code is shown below, extracted from the code synthesized by MESA. Only a partial listing is shown, sufficient to illustrate the single bMSC. Two messages are first defined in the `mtype` block, corresponding to signals in the bMSC. In addition, two channels (message buffers) are defined as instances of type `chan`. The `MS_In` channel is a message queue from which the MS process receives its messages, and to which all other processes can send. Similarly, the BSS process reads from the `BSS_In` channel. Both channels were assigned a length of 10, and no message overflow was found to occur during the simulation. The send and receive operations are translated to Promela's communication mechanism. The ``BSS_In ! chan_req'' string means that the `chan_req` message is immediately sent to the BSS's incoming message buffer. The ``MS_In ? [chan_assign]" defines a read operation from the incoming message queue. The MS first inspects the channel for the presence of the `chan_assign` message, and blocks if it is not found. The MS will resume execution once the `chan_assign` message arrives. Both processes run concurrently and are started in a separate initialization block called `init`. The `atomic` keyword defines a block within which no interleaving can occur.

```
mtype = {  chan_req,   /* message for requesting channel */
           chan_assign /* message for assigning channel */
         };

chan Input_MS=[1] of {mtype};   /* channel for messages to MS */
chan Input_BSS=[1] of {mtype};  /* channel for messages to BSS */

proctype MS() {
 State_MS_1_Chan_Req :
  atomic {
   Input_BSS!chan_req;          /* MS requests a channel from BSS */
   Input_MS?[chan_assign] -> Input_MS?chan_assign; }
                                /* MS receives the assigned channel */
}

proctype BSS() {
 State_BSS_1_Chan_Req :
  atomic {
   Input_BSS?[chan_req] -> Input_BSS?chan_req;
                            /* BSS receives channel request from MS */
   Input_MS!chan_assign; }       /* BSS assigns a channel to MS */
}
```

```
init {
 atomic { run BSS(); run MS(); }        /* start running the processes */
}
```

Non-local branching choice

Non-determinism is an inherent feature of MSC's. At the completion of a sequence specified in a single bMSC, it is possible that the successive behavior will occur according to one of several bMSC's. For example, refer to the `Service_req` hMSC in figure 5.4 on the next page. After a channel has been allocated according to the `Chan_Req` bMSC, the MS may initiate a call or answer an incoming call. The bMSC's corresponding to these actions are `Call_Set_Req` and `Pag_Res_Req`, respectively. A necessary condition is that all processes must choose the same alternative flow of control. The problem of non-local branching choice (see Section 4.5.5 on page 110), where the first message sent in multiple successor bMSC's originates from different processes, was avoided altogether whenever possible, in the following manner:

- First, all successor bMSC's of a branching point were specified to have the same process being the first sender, whenever possible. This allows the other processes to use the wait-and-see strategy (see Section 4.5.5 on page 110) to choose their actions depending on the type of message that they first receive from the sender [65].
- It was also necessary to use normalization of bMSC's to ensure that for each branching node bMSC, the successor bMSC's did not share a common prefix of message exchange sequences. Any common prefix in the successors was delegated to the root bMSC of the successors. This is a requirement of the non-local branching choice detection algorithm [59].

The complete control flow of the model could not, however, be implemented using the wait-and-see approach alone. In some cases, it was not possible to specify a unique sender process in the successor bMSC's, as two

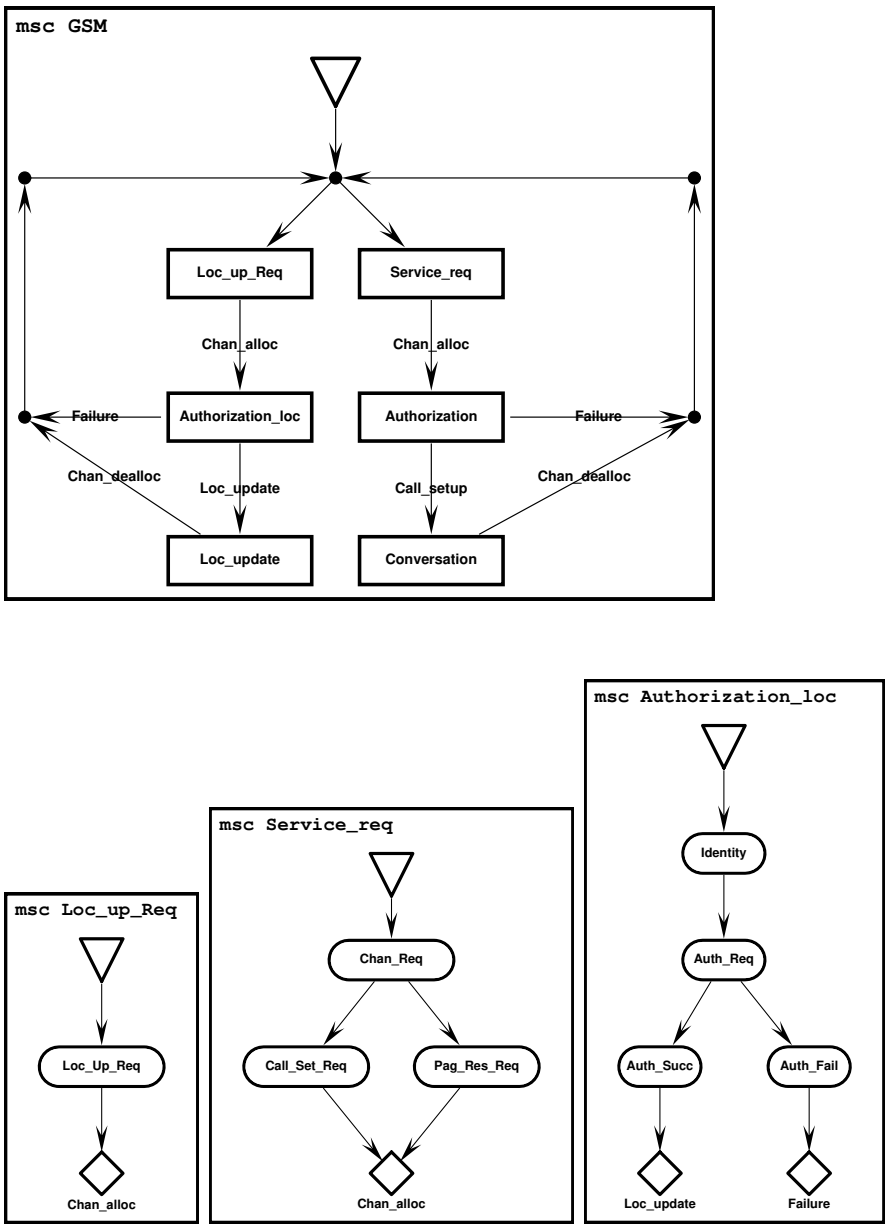

Figure 5.4: hMSC's of the GSM Promela Model (part 1).

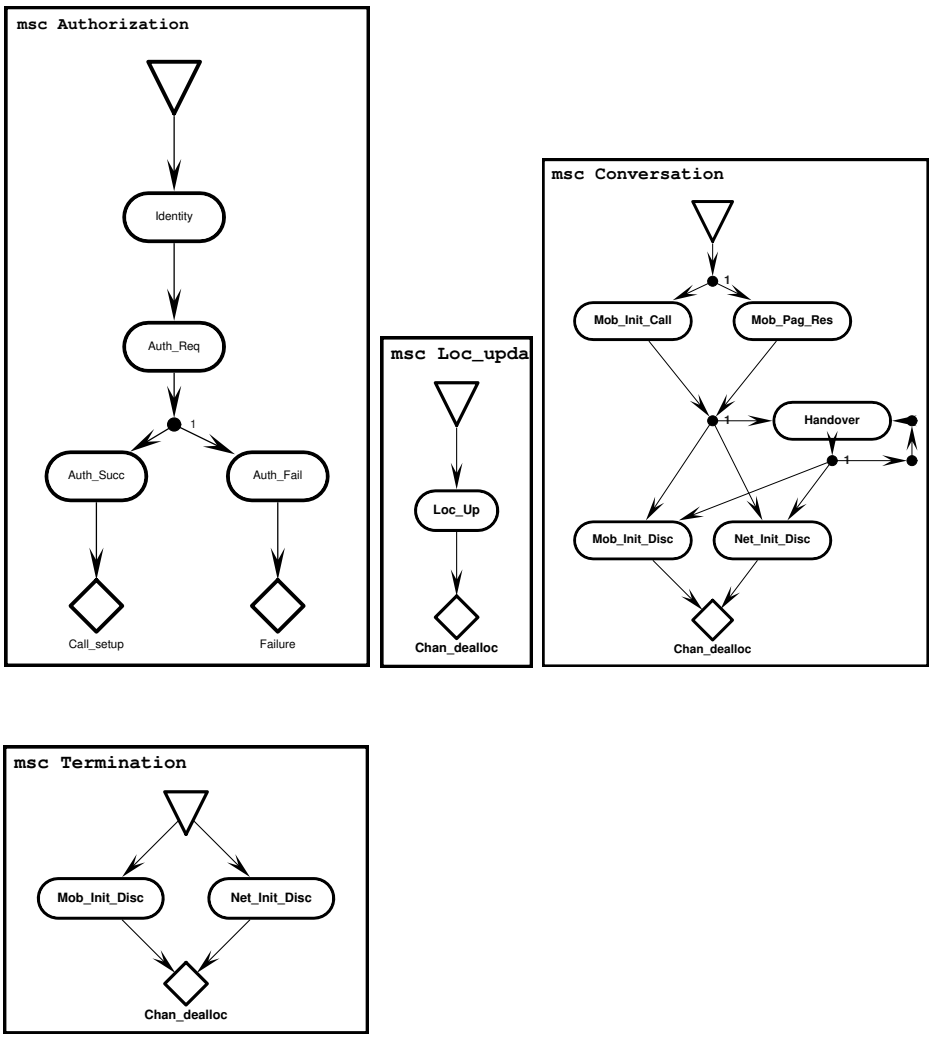

Figure 5.5: hMSC's of the GSM Promela Model (part 2).

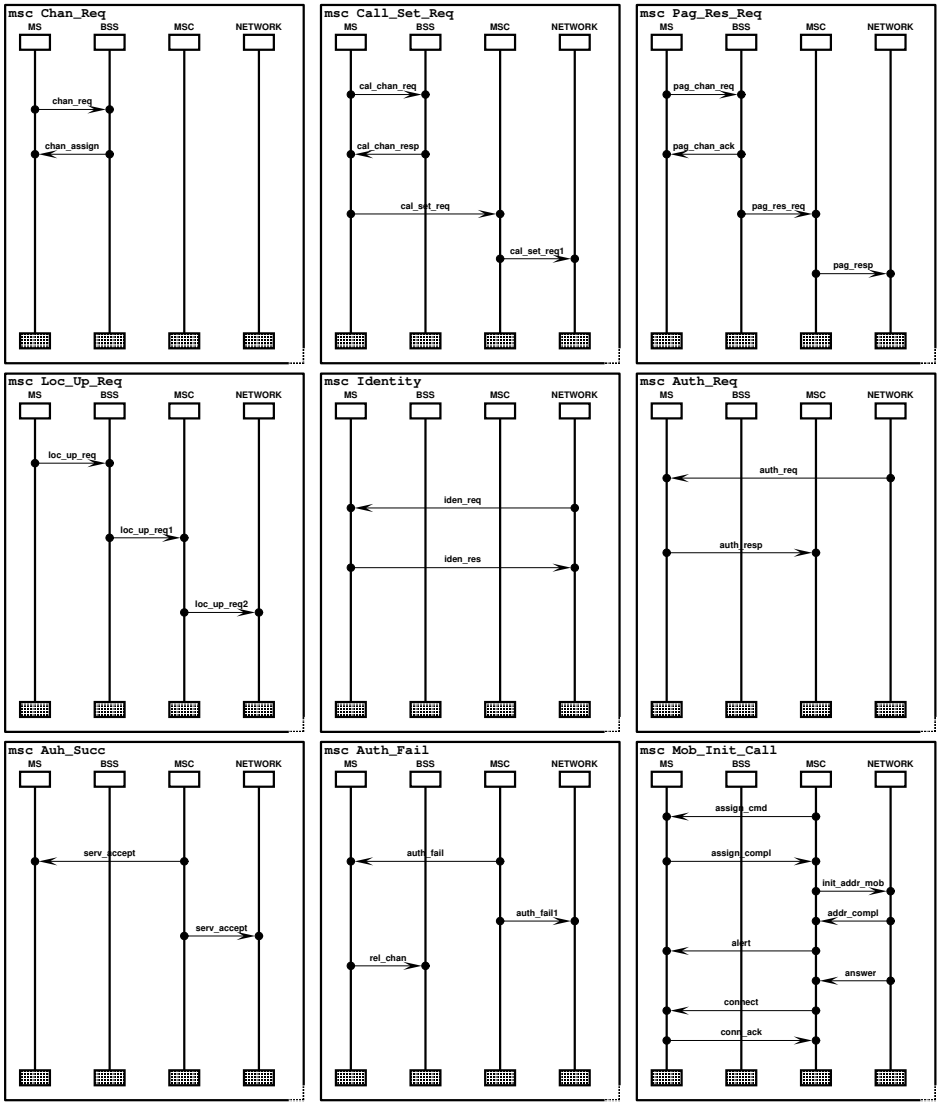

Figure 5.6: bMSC's used in the hMSC's of figure 5.4 for specifying the Promela Model (part 1).

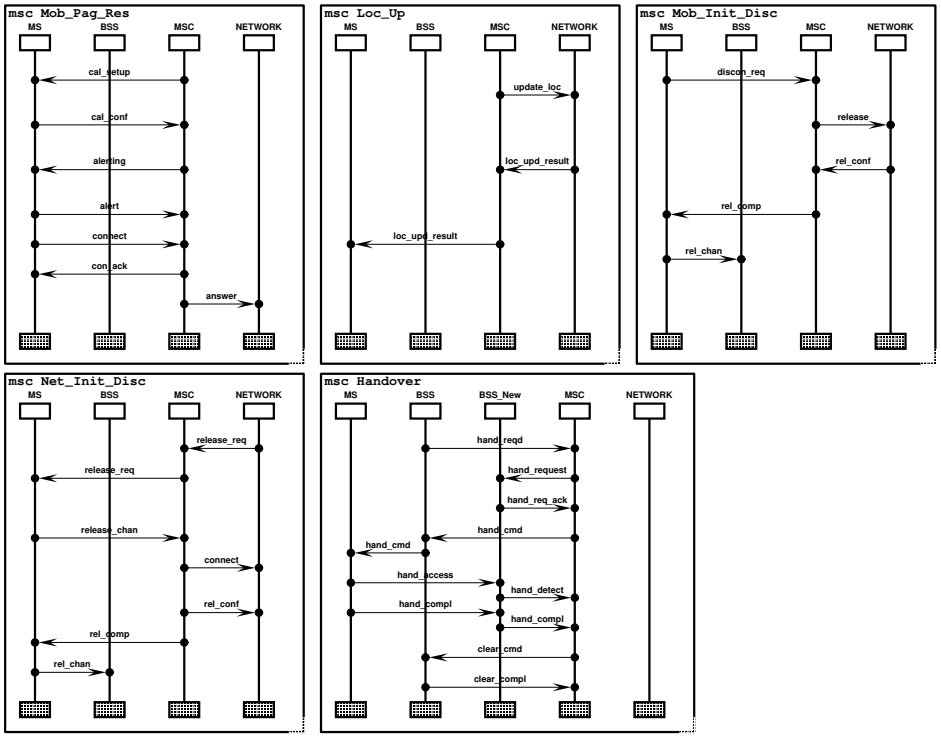

Figure 5.7: bMSC's used in the hMSC's of figure 5.4 for specifying the Promela Model (part 2).

or more processes had to make their choices independently. In the precursory, experimental hand-coded Promela model, global variables were used to ensure a unique branching choice for all processes to resolve non-local choice. One occurrence of the use of a global variable was to record the choice of service type after channel allocation so that after successful authentication, all processes could branch according to that service type. Another occurrence is in the call disconnection phase of a mobile where otherwise no unique sender process in the alternative bMSC's is possible, as either the network or the mobile station can initiate a disconnection request. Each global variable was set at the branching point by a preselected process, read by others, then reset in the `idle` state.

However, the hand-coded global variables proved to be more awkward and restrictive than the synchronizing history variable mechanism (also relying on global variables) supported by the automatic synthesis feature of Mesa 4.5.3. History variables were configured for each non-local branching point.

5.5 Validation of model

This section presents the summarized results of the validation of the Promela model against the high-level requirements, with respect to the Mobility Management features of GSM. The properties of the system requirements which were required to hold in all circumstances were elicited and formalized in Linear Temporal Logic.

LTL (Linear Temporal Logic) is a branch of formal logic used for expressing temporal properties of a system. It makes use of symbols to denote temporal properties [53]: $\Box, \Diamond$ and U represent the *always, eventually* and *until* operator, respectively. p, q, r, s, etc. , are used to represent the state predicates. The symbols && and || denote conjunction and disjunction, respectively. In order to define the above variables in the Promela code, the values of global variables and the labels of the control points of individual processes were used. For example, consider the first high-level requirement of 5.5.1 on page 157. The p and q variables were defined as follows according to the Promela code:

```
p:  MS[1]@State_MS_3_Pag_Res_Req\\
q:  v_ms_chan_assign == 1\\
r:  MS[1]@State_MS_2_Loc_Up_Req
```

By p, the statement label `State_MS_3_Pag_Res_Req` is being identified in the Promela code for the process instance `MS`, corresponding to its state 3. `Pag_Res_Req` is also the name of the original bMSC from which this state was generated. r is a label corresponding to state 2 of the same process. `v_ms_chan_assign` is a control point in the code where the MS is assigned a channel. p, q and r define the truth values of the corresponding logical comparisons. `v_ms_chan_assign` is a global variable of type `byte` in the Promela code. The state labels demonstrate good traceability between the original MSC specification and the Promela code, so that it is easier to track down errors.

Using the SPIN tool, the LTL formula of each high level requirement was converted to a Büchi automaton, also called a *never claim* [50, 51]. This automaton is a specific type of transition system that accepts only those execution traces of the model that satisfy this LTL formula. Using this never claim, SPIN can make a partial or exhaustive search of all system states in order to check whether the formula holds of the model, or not. In other words, SPIN checks whether that property is indeed satisfied or not. See Section 3.3.1 on page 46 for a more thorough discussion of Büchi automata.

5.5.1 High-level requirements and LTL formulæ

The following is a list of high-level requirements representative of Mobility Management, and their encoding in LTL formulæ. The definitions of specific messages are also presented, with reference to the Promela model created. The model was also checked for possible deadlocks and livelocks, and no invalid end states or non-progress cycles were detected.

Deadlock

High-level requirement: The system should not exhibit any deadlock. *Deadlocks* are states in which no further execution is possible, for instance, because all processes are waiting for conditions that can never be fulfilled [50].
Method: The validation of this requirement was performed by using the invalid end states detection feature of Spin. In the GSM model, only the end of the body of a process type declaration was a valid end state. Since the system is event-driven and never terminates, the end of a process body cannot be reached. In transformational systems, a valid end state can be inserted at the location of an end symbol in the MSC specification.

Livelock

High-level requirement: The system should not exhibit any livelock. *Livelocks* are execution sequences that can be repeated infinitely often without ever making effective progress.
Method: The validation of this requirement was done by adding progress labels (see footnote 8 on page 47) to the Promela code and using the non-progress cycles detection feature of Spin. For example, a progress label was used to indicate the allocation of a channel to a mobile station after its request. Livelock is more applicable, however, to situations where a cycle exists in the MSC specification that can be executed infinitely often such that a condition is not satisfied. For instance, the handover operation specified in the `Handover` bMSC can occur infinitely often so that the connection is never terminated. However, this scenario would entail constant and rapid physical relocation of the mobile across base station boundaries, and is extremely unlikely.

Power off

High-level requirement: If the MS has not been granted a channel to the BSS, then it cannot respond to any paging request.
Formula: $\Box((\neg pUq)||(r))$
p = Respond to a paging request. (`pag_chan_req` message in `Pag_Res_Req` bMSC)
q = The MS has been granted a channel to the BSS. (`chan_assign` message in `Chan_Req` bMSC)
r = The MS performs a location update request. (`loc_up_req` in `Loc_Up_Req` bMSC)

Network connection

High-level requirement: When the MS is turned on, it is eventually granted a channel from the BSS.
Formula: $\Box(p \rightarrow \Diamond q)$
p = The MS is turned on. (`chan_req` in `Chan_Req` bMSC)
q = The MS is granted a channel from the BSS. (`chan_assign` in `Chan_Req` bMSC)

Service Request

High-level requirement: If the MS is idle, then it will eventually initiate a call setup, a paging response, or a location update.
Formula: $\Box(p \rightarrow \Diamond(q \mid\mid r \mid\mid s))$
p = The MS is idle. (after `chan_assign` in `Chan_Req` bMSC)
q = The MS initiates a call setup. (`cal_chan_req` in `Call_Set_Req` bMSC)
r = The MS initiates responds to a paging request. (pag_chan_req in `Pag-_Res_Req` bMSC)
s = The MS initiates a location update. (`loc_up_req` in `Loc_Up_Req` bMSC)

Identification

High-level requirement: After the MS initiates a new service request, it must eventually be identified by the network.
Formula: $\Box((q \mid\mid r \mid\mid s) \rightarrow \Diamond t)$
q = The MS initiates a call setup. (`cal_chan_req` in `Call_Set_Req` bMSC)
r = The MS initiates responds to a paging request. (`pag_chan_req` in `Pag-_Res_Req` bMSC)
s = The MS initiates a location update. (`loc_up_req` in `Loc_Up_Req bMSC`)
t = The MS is identified by the network. (`iden_req` in `Identity` bMSC)

Authentication

High-level requirement: After being identified, the MS must be authenticated by the network, and the result is a success or failure.
Formula: $\Box(p \rightarrow \Diamond(q \mid\mid r))$
p = The MS is identified. (`iden_req` in `Identity` bMSC)
q = The authentication is successful. (`serv_accept` in `Auth_Succ` bMSC)
r = The authentication is unsuccessful. (`auth_fail` in `Auth_Fail` bMSC)

Successful authentication

High-level requirement: The MS can only connect to another party if it is first authenticated.
Formula: $\Box(p \rightarrow q)$
q = The MS is successfully authenticated. (`serv_accept` in `Auth_Succ bMSC`)
p = The MS is connected to another party (the call is initiated by the former or latter). (`cal_setup` in `Mob_Pag_Res` bMSC or `assign_cmd` in `Mob_Init_Call` bMSC)

Authentication failure

High-level requirement: If the authentication fails, then the mobile releases the call channel, and can attempt to request a channel from the BSS again.
Formula: $\Box(p \rightarrow (q \rightarrow \Diamond r))$
p = The authentication of the MS fails. (`auth_fail` in `Auth_Fail` bMSC)
q = The call channel is released by the MS. (`rel_chan` in `Auth_Fail` bMSC)
r = The MS requests a channel to the BSS. (`chan_req` in `Chan_Req` bMSC)

Location update

High-level requirement: If the MS moves to a new location area, it remains connected to the network.
Formula: $\Box(p \rightarrow q)$
p = The MS has updated the network of its new location. (`loc_upd_result` in `Loc_Up` bMSC)
q = The MS remains connected to the network. (`chan_assign` in `Chan_Req` bMSC)

Conversation

High-level requirement: After a successful authentication, if the MS initiated a call or answered a page, then it will end up in the connected state.
Formula: $(\Box(q \&\& r) \,||\, \Box(p \&\& r)) \rightarrow \Diamond s$
q = The MS has requested a paging response. (`pag_chan_req` in `Pag_Res_Req` bMSC)
r = The MS is successfully authenticated. (`serv_accept` in `Auth_Succ` bMSC)
p = The MS has requested a call setup. (`cal_chan_req` in `Call_Set_Req` bMSC)
s = The MS is in the connected state with another party. (`assign_cmd` in `Mob_Init_Call` bMSC or `cal_setup` in `Mob_Pag_Res` bMSC)

Disconnection

High-level requirement: Either the caller or the callee can disconnect a call.
Formula: $\Box((p \mid\mid q) \rightarrow \Diamond r)$
p = The network initiates a disconnection. (`release_req` in `Net_Init_Disc` bMSC)
q = The MS initiates a disconnection. (`discon_req` in `Mob_Init_Disc` bMSC)
r =The call is terminated. (`chan_req` in `Chan_Req` bMSC)

Channel release

High-level requirement: Upon termination of the connection, the call channel is released, and the MS can now initiate a new service request.
Formula: $(\Box(p \rightarrow s) \mid\mid \Box(q \rightarrow t)) \rightarrow \Diamond r$
p = The network initiates a disconnection. (`release_req` in `Net_Init_Disc` bMSC)
q = The MS initiates a disconnection. (`discon_req` in `Mob_Init_Disc` bMSC)
s = The channel is released. (`rel_chan` in `Net_Init_Disc` bMSC)
t = The channel is released. (`rel_chan` in `Mob_Init_Disc` bMSC)
r = The MS reconnects to the network and can initiate a new service request. (`chan_assign` in `Chan_Req` bMSC)

Handover

I. High-level requirement: If handover is required, then the MS will be able to connect to a new base station.
Formula: $\Box(((p\&\&q) \rightarrow \Diamond r) \mid\mid ((p\&\&r) \rightarrow \Diamond q))$
p = Handover is required. (`hand_reqd` in Handover)
q = The MS is currently connected to BSS 1.
r = The MS is currently connected to BSS 2.

II. High-level requirement: If the MS is in conversation with another party, and handover occurs, then the MS will continue its conversation without interruption.
Formula: $\Box(p \rightarrow \Diamond s)$
p = Handover is required. (hand_reqd in Handover)
s = The MS is in the conversation state. (after `hand_compl` in `Handover`)

5.5.2 Summary of validation results

All of the properties of the high-level requirements that were chosen were found to be valid (in the portion of the state space covered) in the Promela model using the SPIN tool. The methodology used is quite practical in terms of the computing resources required to analyze the model. *Supertrace* state space exploration was always performed[1]. Statistics for the validation runs are shown in table 5.3. For comparison, the statistics for the earlier hand-coded model are shown, too, in table 5.4. The latter required significantly fewer resources, because history variables were not used to resolve non-local choice. Hence, exhaustive state exploration was performed on the hand-coded model. However, the model was less flexible in that the synchronization through global variables resulted in the processes running

[1]Supertrace is a controlled partial-search technique using hashing to perform the largest possible search within a set amount of memory [50]. Due to the possibility of unresolved hash conflicts, 100% coverage cannot be guaranteed. *Supertrace* is a controlled partial-search technique that is only meant for the validation of protocol systems that cannot be analyzed exhaustively, which is the case here, and usually with any reasonably complex system. It cannot be guaranteed that the tests would still be valid if an exhaustive search of the state space was possible and performed.

more in lock-step with each other, and the same process always decided on non-local branching decisions. Although it was more optimized, the hand-coded model required considerable time and patience to encode, while the generated model was produced by MESA in seconds.

5.6 Conclusions

A step-by-step procedure has been shown for requirements specification and validation of the Mobility Management aspect of GSM using Promela/SPIN. Message Sequence Charts have been used for the early stage of requirements specification and to create the operational Promela model. A representative set of high-level requirements were elicited from the GSM specification standard and coded in Linear Temporal Logic in order to prove a number of properties to hold true of the Promela model. The results obtained for the validation of the Promela model were also discussed. The procedure shown is easy to follow and effective for the specification and validation of a complex system like the Mobility Management service of GSM. A complicated real-world communications protocol has been modeled. The approach demonstrated allows for an easy and complete analysis of a subset of almost any communications protocol.

The Promela language was found to be an appropriate method of modeling Mobility Management in GSM. The high degree of message interaction in the system lent itself to Promela's support for a message-passing model, and the entities of the GSM system were easily modeled using multiple process instances in Promela. The resulting model was automatically validated with the potential for easy modification if necessary. Although the elicited set of requirements based on the standard were not found to have any inherent faults in the final model, a number of minor errors were detected and subsequently corrected in the simplified operational model during development. The principal causes of them were missing messages eliminated due to oversimplification. These errors were easily detected through SPIN's validation features.

The approach used was highly automated, simple, quick, and effective. The MSC specification of the system was quickly entered using the editing

Generated model	
Processor	AMD Athlon^{TM}700 MHz
Memory	256 MB RAM
Duration of deadlock test	16 min.
Maximum depth	$8 \cdot 10^6$ states
No. of state transitions	$3.98 \cdot 10^6$
No. of atomic steps	$4.66 \cdot 10^6$
Total memory usage	251 MB
Average duration of LTL tests	4 min.
Maximum memory usage	252 MB
Maximum depth	$2 \cdot 10^6$ states

Table 5.3: Statistics for validation of LTL tests on GSM model generated using MESA. The total number of transitions and atomic steps required for the LTL tests varied greatly depending on the high-level requirements tested.

Hand-coded model	
Processor	UltraSPARC IIi^{TM}300 MHz
Memory	256 MB RAM
Duration of deadlock test	15 sec.
No. of state transitions	$21.5 \cdot 10^3$
No. of atomic steps	$5.73 \cdot 10^3$
Total memory usage	3.6 MB
Average duration of LTL tests	11 sec.
Maximum memory usage	3.6 MB
Maximum no. of transitions	$21.5 \cdot 10^3$
Maximum no. of atomic steps	$5.76 \cdot 10^3$

Table 5.4: Statistics for validation of LTL tests on hand-coded GSM model.

facilities of MESA. Syntactic and static-based analysis, such as identification of non-local choice and Z.120 compliance, was automatically performed inside the tool. History variables were easily added to the appropriate branching points. A Promela model of the system was automatically synthesized using MESA. After some minor work in adding progress labels and state variables to select portions of the code, the LTL tests were easily entered and automatically verified in seconds or minutes using the SPIN tool, with the exact location of any problems displayed on a progress chart for debugging.

In spite of the completeness of the described method, one area not supported by the Promela language is the notion of incorporating real time constraints into the model. This limitation of Promela prevented the modeling of some parts of the specification related to the exact timing requirements. One example is the mechanism by which the MS regularly polls the received signal strengths for all surrounding cells at least once every 6 seconds for purposes of activating handover in the case of an out-of-bounds value. Modeling real-time constraints is not the focus of this thesis, however, and, in any case, this requirement can be abstracted away.

Chapter 6

ROOM Synthesis of GSM Model

The GSM model of the previous chapter was synthesized into a ROOM model, using the proposed features of hierarchical actor structure, replication, and automatic coordination, from chapter 4. This illustrates how an MSC specification can be validated (through Promela synthesis and LTL tests), then synthesized into a design-time ROOM model.

The topmost `Environment` actor is shown in figure 6.1. The internal structure of the GSM subsystem appears in figure 6.2. The finite state machine of the base station (BSS) actor is shown in figure 6.3. The internal behavior of the `Conversation` superstate of the BSS actor appears in figure 6.4. For comparison, the finite state machine of the mobile (MS) actor is shown in figure 6.5.

The base station is modeled as a replicated actor (with a replication factor of 2), so that handover between two BSS's can be simulated. In the BSS actor, the handover operation, from the view of the `BSS` process instance in the original MSC specification, is performed within the `Handover` sub-state of the `Conversation` state. Once handover completes, the original base station (specifically, the channel used) is free to be used as part of another call. Thus, the `CoIn` transition is taken to return to the `Initialization` state, from which a new service can be started. The

base station to which the mobile is handed over to, begins from the opposite end: the `Initialization` state. Upon receiving a handover request (`Hand_Request`) from the MSC, it participates in the processing of the handover operation in the `Handover_Conv` state, from the view of the `BSS_New` process instance in the original MSC specification. It then travels via the `HaCo` transition to the `Handover` substate, and becomes the new base station of the current call. Handover can theoretically occur from one base station to the other an unlimited number of times. All other actors communicate with the correct BSS instance through an indexed array of pointers to the two BSS actor references, and must keep track of the current index number. In comparison, the MS actor does not possess a `Handover_Conv` state in its behavioral description, as the handover process does not terminate a call from its point of view, and is handled entirely within the `Handover` substate of the `Conversation` state.

The `Coordinator` is involved in coordinating non-local branching decisions, such as deciding whether a call request or a paging response scenario should be executed, or whether the mobile authorization succeeds or fails.

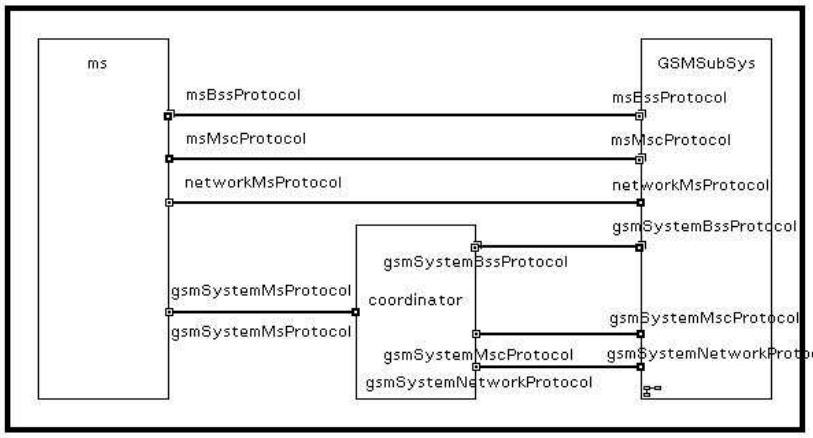

Figure 6.1: The structural diagram of the topmost `Environment` actor of the GSM ROOM model is shown. The mobile is represented by the `ms` process, and interfaces with the `GSMSubSystem` hierarchical actor containing all GSM hardware components. The automatic coordinator is represented by the `coordinator` actor in the middle.

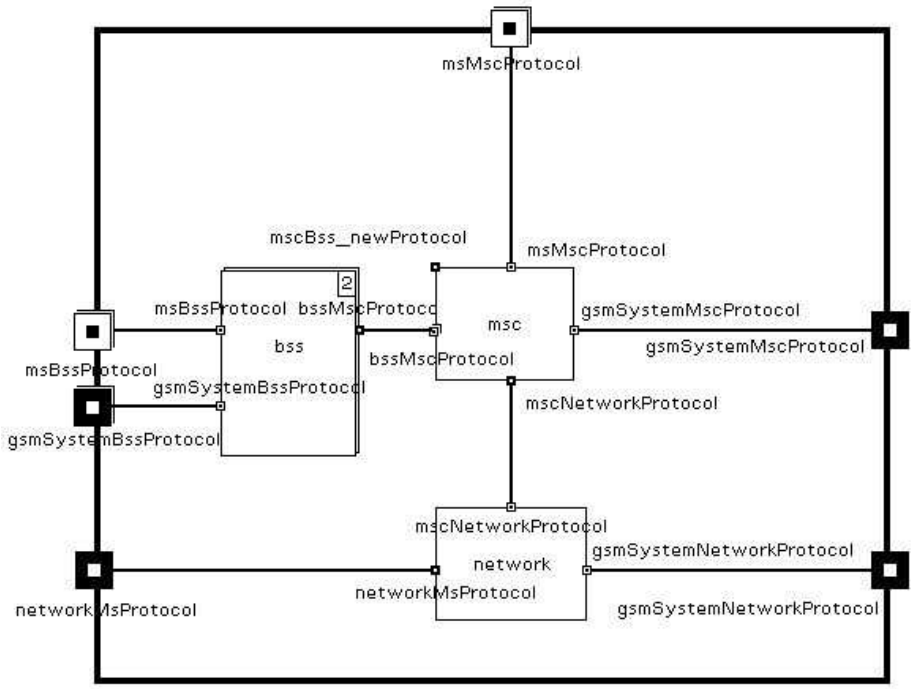

Figure 6.2: The structural diagram of the interior of the `GSMSubSystem` hierarchical actor is shown. The BSS, MSC, and Network components are each represented by their own actor instance. The `BSS` actor is replicated twice so that handover between two base stations can be modeled. The end and relay ports on its interfaces are replicated as well, with a replication factor of 2. Each actor interfaces with the coordinator in the `Environment` actor through the `gsmSystem` protocol.

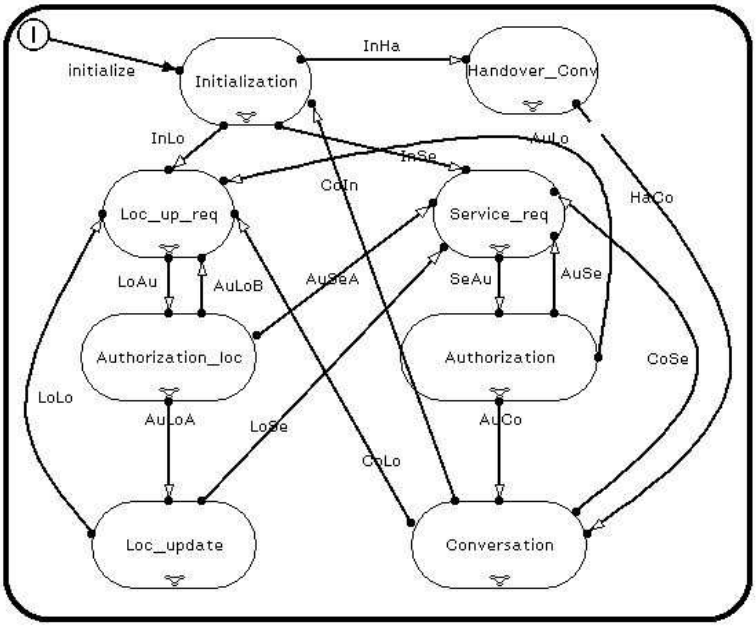

Figure 6.3: The finite state machine of the BSS actor is shown. The `Initialization` state simply serves to select an initial branch: a location update or service request. The branch is selected by the coordinator.

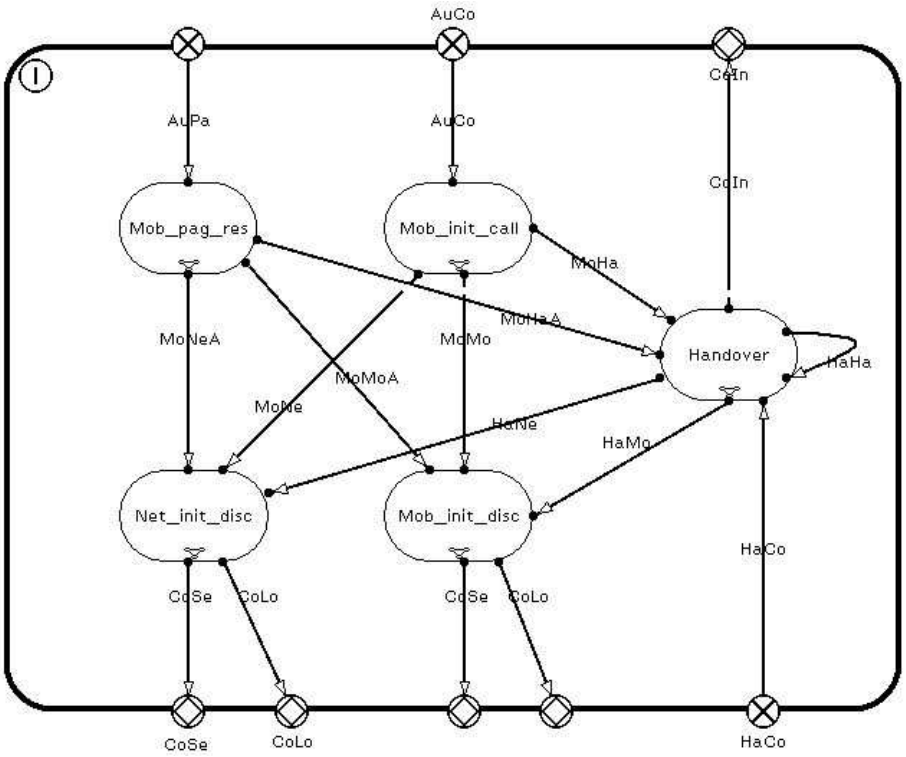

Figure 6.4: The internal state machine of the `Conversation` state of the BSS actor is shown.

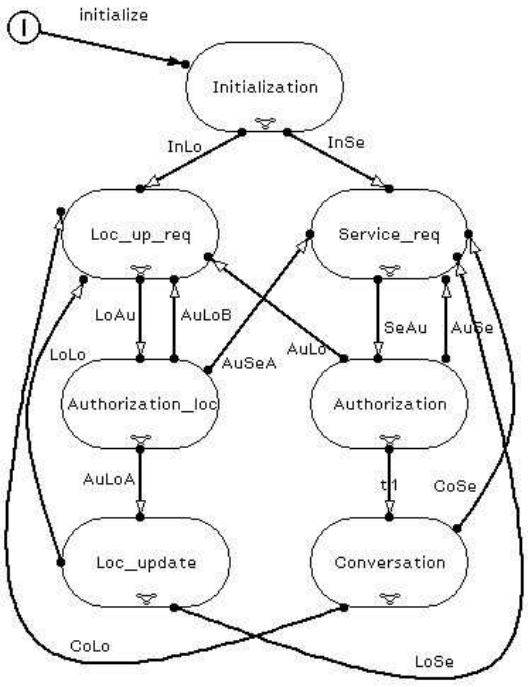

Figure 6.5: The finite state machine of the `MS` actor.

Chapter 7

Final conclusions

7.1 The purpose and process of synthesis

Formal requirements specification at the early stages of complex software systems development helps to reduce errors before they become very expensive to repair. Message sequence charts are a popular method of specifying protocols in real-time systems. The automatic synthesis of models from MSC's reduces errors in the design process. The purpose of MSC synthesis is two-fold:

- To produce a working skeletal prototype for later refinement. It is important for the synthesis result to be of practical value to an applications programmer not necessarily skilled in the practice of formal requirements engineering, and various criteria have been identified. ROOM is an appropriate methodology as a synthesis target due to its wide use in the telecommunications industry and ObjecTime Developer tool support. Although prior work was done in this area, the synthesized model suffered from a lack of structural object-orientation, a lack of traceability in the behavioral view, a lack of support for certain types of refinement and scenarios, and could not be executed without the user's input. A solution to each of these problems has been proposed and verified to work correctly on a POTS telephony system, including hierarchical actor structure, replication, and a global

co-ordination system.

- To produce a model to validate properties from the requirements specification that cannot be directly accomplished with MSC's. The most important concept not present in the MSC standard is an expression of liveness properties. The synthesis of a Promela model from the MSC specification allows these properties to be specified in LTL and validated to determine whether they hold true of the model. Although prior work was done on Promela synthesis algorithms, the result generated a large statespace that could not be practically validated to a reasonable degree. A number of optimizations in the approach and algorithms have been proposed in this thesis, and a complex telecommunications system, the Mobility Management service of GSM, was successfully specified, synthesized, and validated using the MESA and SPIN CASE tools.

The synthesis work presented in this thesis is a step towards the ultimate goal of the automation of the software development process and elimination of errors.

The development process suggested based on the results of the thesis is illustrated in figure 7.1. The synthesis process entails the creation of a working design-time prototype from initial, informally-specified requirements elicited from the customer. After entry of the MSC model and verification of syntactic properties such as connectedness using the MESA toolset, a ROOM model is synthesized and loaded into the ObjecTime Developer environment for refinement, adding design details such as data processing to the state machines. A simulation can then be run to test functional aspects such as timing constraints. To ensure robustness, a Promela model may be synthesized from the MSC specification, and safety and liveness requirements may be translated into LTL properties that are validated against it. After any necessary corrections to the MSC specification, the ROOM model may be re-generated. The process can be followed in an iterative fashion — if the ROOM model is found to exhibit undesired properties at any point requiring modification to the MSC specification, it can be re-synthesized. This process was followed in chapter 5, where a GSM system was synthesized into a Promela model to validate a number of properties, then re-synthesized into

a design-time ROOM model to proceed into the implementation phase.

7.2 Future work

The current process structure in ROOM synthesis is static. Dynamic process creation and destruction is not supported. The addition of these features would expand the spectrum of systems that could be synthesized.

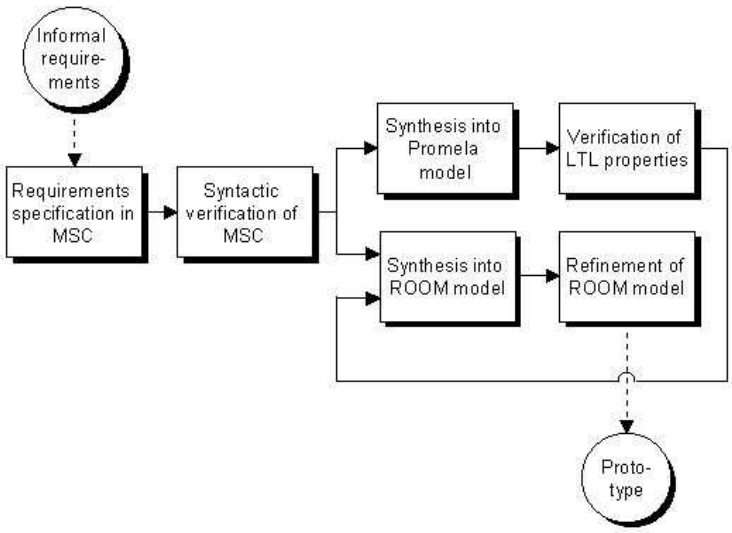

Figure 7.1: The synthesis process.

Appendix A

ROOM Linear Form

The following is an example of the ROOM linear form syntax. It contains the actor definition for the hierarchical actor `PhoneActor_CallBased` in figure 4.14 on page 80. It shows how the structure of an actor is encoded into a file that is imported into OTD.

The rules and syntax of the linear form format were not publicly documented by ObjecTime other than a BNF (Backaus-Naur Form) grammar. The coding of the linear form generation in [5] required a reverse engineering of the format by careful inspection of the linear form exported from manually encoded OTD projects.

The linear form file includes the structural definition of actors, ports, and the system-level package. It includes the specification of instances of class types, replication factors, conjugation of ports, transitions between states in the finite state machine, and associated triggers and actions.

```
VERSION '5.2'

ACTOR CLASS PhoneActor_CallBased
    DESCRIPTION 'Phone process with call initiation and
        call reception components as contained sub-actors.'
    INTERFACE {
        PORTS {
```

```
        DEFINE envPhoneProtocol_A
            ISA CONJUGATED EnvironmentPhoneProtocol;
        DEFINE envPhoneProtocol_B
            ISA CONJUGATED EnvironmentPhoneProtocol;
        }
    }
IMPLEMENTATION {
    STRUCTURE {
        COMPONENTS {
            DEFINE callInit
                ISA SUBSTITUTABLE CallInit;
            DEFINE callRecv
                ISA SUBSTITUTABLE CallRecv;
            }
        BINDINGS {
            DEFINE B1 [1]
                BETWEEN phonePhoneProtocol/callRecv
                AND phonePhoneProtocol/callInit;
            DEFINE B2 [1]
                BETWEEN envPhoneProtocol/callInit
                AND envPhoneProtocol_A;
            DEFINE B3 [1]
                BETWEEN envPhoneProtocol/callRecv
                AND envPhoneProtocol_B;
            }
        }
    BEHAVIOR {
        LANGUAGE 'C++'
        FSM
        }
    };
```

Bibliography

[1] C. Ghezzi, M. Jazareyi, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, New Jersey, 1991.

[2] A. Mehrotra. *GSM System Engineering*. Artech House, Boston, 1997.

[3] B. Selic, G. Gullekson, P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.

[4] M. Shaw, D. Garlan. *Software Architecture*. Prentice Hall, New Jersey, 1996.

[5] S. Leue, L. Mehrmann, M. Rezai. "Synthesizing ROOM Models from Message Sequence Chart Specifications." Technical Report 98-06, Dept. of Electrical & Computer Engineering, University of Waterloo, 1998.

[6] S. Leue, L. Mehrmann, M. Rezai. "Synthesizing ROOM Models from Message Sequence Chart Specifications." 13th IEEE Conference on Automated Software Engineering, Honolulu, Hawaii, October 1998.

[7] S. McConnell. *Rapid Development*. Microsoft Press, Redmond, 1996.

[8] E. Rudolph, P. Graubmann, J. Grabowski. "Tutorial on Message Sequence Charts."

[9] D. Harel, H. Kugler. "Synthesizing Object Systems from LSC Specifications." The Weizmann Institute of Science, Rehovot, Israel, 1999.

[10] D. Harel. "Statecharts: A Visual Approach to Complex Systems." 1984.

[11] B. Selic. "An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems." 1993.

[12] ObjecTime Ltd. *ObjecTime Developer User Guide*. Product release 5.2, document version 1.0, 1998.

[13] G. Gullekson, B. Selic. "Design Patterns for Real-Time Software." Embedded Systems Conference West, 1996.

[14] W. Damm, D. Harel. "LSCs: Breathing Life into Message Sequence Charts." Technical Report CS98-09, Weizmann Institute Tech., April 1998.

[15] D. Harel, E. Gery, "Executable Object Modeling with Statecharts." Proceedings of the 18th International Conference on Software Engineering, IEEE Press, March 1996.

[16] D. Harel, E. Gery, "Executable Object Modeling with Statecharts." (Revised version). IEEE Computer, 1997.

[17] G. Holzmann. "Tutorial: Design and Validation of Protocols." AT&T Bell Laboratories.

[18] D. Harel, M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, New York, 1998.

[19] "Specification and Description Language (SDL) Tutorial." Telelogic, 2000.

[20] S. Mauw. "The Formalization of Message Sequence Charts."

[21] H. Ben-Abdallah, S. Leue, "MESA: Support for Scenario-Based Design of Concurrent Systems."Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, 1998. March/April 1998, Vol. 1384 of *Lecture Notes in Computer Science,* p. 118 - 135, Springer Verlag, 1998.

[22] A. Engels, S. Mauw, M. Reniers. "A Hierarchy of Communication Models for Message Sequence Charts." Report 97/11, Dept. of Computer Science, Eindhoven University of Technology, 1997.

[23] D. Brand, P. Zafiropulo. "On Communicating Finite State Machines." Journal of the ACM, 1983.

[24] D. Harel, O. Kupferman. "On the Inheritance of State-Based Object Behavior." Technical Report MCS99-12, The Weizmann Institute of Science, Rehovot, Israel.

[25] F. Belina, D. Hogrefe, A. Sarma. *SDL with Applications from Protocol Specification.* Prentice-Hall, Hemel Hempstead, Hertfordshire, England, 1991.

[26] Morris. *Metrics for Object-Oriented Software Development Environments.* Master's Thesis, M.I.T. Sloan Schoool of Management, 1989.

[27] Chidamber and Kemerer. "A Metrics Suite for Object-Oriented Design." IEEE Trans. Software Eng., vol. 20, no. 6, June, 1994.

[28] Institute of Electrical & Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology.* ANSI/IEEE Standard 729-1983. New York, 1983.

[29] P. Tavolato and K. Vincena. "A Prototyping Methodology and Its Tool." In *Approaches to Prototyping,* R. Budde et al., eds., Berlin: Springer-Verlag, 1984.

[30] G. Coulouris, J. Dollimore, T. Kindberg. "Distributed Systems: Concepts and Design, Second Edition." Addison-Wesley, Harlow, England, 1996.

[31] ObjecTime Ltd. http://www.objectime.com

[32] Rational Software et al. "UML Summary: Version 1.1." September 1997.

[33] Rational Software et al. "UML Notation Guide: Version 1.1." September 1997.

[34] S. Brinkkemper, S. Hong, A. Bulthuis, G. van der Goor. "Object-Oriented Analysis and Design Methods: A Comparative Review." University of Twente.

[35] CCITT. *CCITT Recommendation Z.120: Message Sequence Chart (MSC)*. CCITT, Geneva, 1992.

[36] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, April, 1996.

[37] SDL Forum Society. *MSC-2000: SDL Forum Version of Z.120*. SDL Forum Society, Darmstadt, Germany.

[38] J. Cooper. "JavaDesign Patterns: a Tutorial." 2000.

[39] S. Leue. "Methods and Semantics for Telecommunications Systems Engineering." Doctoral Dissertation, University of Berne, Dec. 1994.

[40] P. Ladkin, S. Leue, *Four issues concerning the semantics of Message Flow Graphs.* Proceedings of the 7th International Conference on Formal Description Techniques, Chapman & Hall, 1995.

[41] S. Alhir. "UML In A Nutshell." O'Reilly, Sebastopol, CA, 1998.

[42] M. Fowler, K. Scott. "UML Distilled: Second Edition." Addison-Wesley, Reading, MA, 2000.

[43] G. Booch, J. Rumbaugh, I. Jacobson. "The Unified Modeling Language User Guide." Addison-Wesley, Reading, MA, 1999.

[44] J. Rumbaugh et al. "Object-Oriented Modeling and Design." Prentice Hall, New Jersey, 1991.

[45] G. Booch. "Object-Oriented Analysis and Design With Applications: Second Edition." Benjamin/Cummings, Redwood City, CA, 1994.

[46] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Proceedings of the 3rd IFIP Conf. on Formal Methods for Open-Object-Based Distributed Systems (FMOODS '99)*, (P. Ciancarini, A. Fantechi and R. Govrieri, eds.), Kluwer Academic Publishers, pp. 293-312, 1999.

[47] A. M. Davis, *Software requirements: objects, functions, and states,* Upper Saddle River, USA, Prentice Hall, 1993.

[48] ETSI, *GSM: Digital Cellular Telecommunications System (Phase 2+); Handover Procedures (GSM 03.09),* Valbonne, France, ETSI, August, 1997.

[49] *GSM: Digital Cellular Telecommunications System (Phase 2+); General Description of a GSM Public Land Mobile Network (PLMN) (GSM 01.02),* Valbonne, France, ETSI, 1996.

[50] Gerard J. Holzmann, *Design and validation of computer protocols,* Englewood Cliffs, N.J., Toronto, Prentice Hall, 1991.

[51] Gerard J. Holzmann, "The Model Checker Spin," *IEEE Trans. on Software Engineering,* 23(5), May 1997, pp. 279-295.

[52] S. Leue, "QOS specification based on SDL/MSC and temporal logic," In G. v. Bochmann, J. de Meer, and A. Vogel, Editors, *Proceedings of Workshops on Distributed and Multimedia Applications and Quality of Service Verification,* Montreal, Quebec, Canada, May 1994.

[53] Zohar Manna and Amir Pnueli, *The Temporal logic of reactive and concurrent systems,* New York, Springer-Verlag, 1992.

[54] Moe Rahnema, "Overview of the GSM system and protocol architecture," *IEEE Communications Magazine,* April, 1993.

[55] ITU-T, Recommendation Z.120, Message Sequence Charts, ITU-Telecommunication Standardization Sector, Geneva, Switzerland, November 1999.

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

[57] A. Mehrotra, *GSM System Engineering,* Artech House, Boston, London, 1997.

[58] John Scourias, "A Brief Overview of GSM," *http://kbs.cs.tu-berlin.de/ jutta/gsm/js-intro.html.*

[59] H. Ben-Abdallah and S. Leue, "Syntactic Detection of Process Divergence and Non-Local Choice in Message Sequence Charts," in: E.

Brinksma, (ed.), *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems TACAS '97,* Enschede, The Netherlands, April 1997, *Lecture Notes in Computer Science,* Vol. 1217, p. 259 - 274 Springer-Verlag, 1997.

[60] F. Orava and J. Parrow, "An algebraic verification of a mobile network," *Formal Aspects of Computing,* 497-543, 1992.

[61] S. L. Pfleeger, "Software Engineering: Theory and Practice." Prentice-Hall, Upper Saddle River, USA, 1998.

[62] Bjorn Victor, Faron Moller, "The Mobility Workbench - A Tool for the π Calculus," *Lecture Notes in Computer Science,* vol. 818, pp. 428 - 440, Springer Verlag, 1994.

[63] L. Fredlunc and Fredrik Orava, "Modeling dynamic communication structures in LOTOS," In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV,* pp. 185-200, North-Holland, Amsterdam, Netherlands, November 1991.

[64] H. Garavel and J. Sifakis, "Compilation and verification of LOTOS specifications," *Proc. of 10th International Symposium on Protocol Specification, Testing and Verification (PSTV),* Ottawa, IFIP, 1990.

[65] S. Leue and P.B. Ladkin, "Implementing and Verifying MSC Specifications Using Promela/XSpin," In: J.-C. Gregoire, G. Holzmann, and D. Peled (eds.), *Proceedings of the DIMACS Workshop SPIN '96, the 2nd International Workshop on the SPIN Verification System DIMACS, Series,* Volume 32, American Mathematical Society, Providence, R.I., 1997.

[66] Ø. Haugen. "MSC-2000 Interaction for the New Millenium," Ericsson NorARC.

Appendix B

Glossary

Call origination. The initiation of a call request from one phone to another.

Call termination. The response to an incoming call request.

Caller. The phone initiating a call request.

Callee. The phone that is the recipient of a call request.

ISDN. Integrated Services Digital Network. An international communications standard for sending voice, video, and data over digital telephone lines or normal telephone wires.

POTS. Plain Old Telephone Service. The standard analogue voice-based phone system that most homes use.

TTRX. Touch-tone receiver. A device that senses which digit is being dialed.