Building a Tool for Synthesis of Correct Design from Interaction Specifications

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Praveen N. Moorthy

Committee in charge:

Professor Ingolf H. Krueger, Chair
Professor William G. Griswold
Professor Ranjit Jhala

2006
Copyright
Praveen N. Moorthy, 2006
All rights reserved.
The thesis of Praveen N. Moorthy is approved:

Chair

University of California, San Diego

2006
# TABLE OF CONTENTS

Signature Page ......................................................... iii  
Table of Contents ..................................................... iv  
List of Figures ........................................................ vii  
List of Tables .......................................................... ix  
Abstract ................................................................. x  

I  Introduction ......................................................... 1  
A. Problem Statement ................................................ 1  
B. Model Driven Design .............................................. 2  
   1. What are these models? ...................................... 3  
C. Shortfalls in Tool Chains ....................................... 8  
D. Proposed Solution ............................................... 8  
E. Example ............................................................. 10  
F. Contributions and Outline ...................................... 12  

II  Methodology and Notation ........................................ 14  
A. Introduction ....................................................... 14  
B. Methodology ....................................................... 14  
   1. Requirements ................................................ 15  
   2. Component Identification .................................. 15  
   3. Interaction Specification .................................. 16  
   4. System Structure and State Machines .................... 16  
   5. Implementation ............................................. 18  
C. Notation ............................................................ 18  
   1. Message Sequence Charts ................................... 18  
   2. State Machines .............................................. 25  

III  Tool Requirements ............................................... 26  
A. General Overview ............................................... 26  
B. User Interface Requirements .................................. 27  
C. Translation Requirements ...................................... 29  
D. Import Export Requirements .................................. 29  

IV  Design Rationale, Decisions and Consequences ............... 31  
A. Design Rationale ............................................... 31  
B. Decisions and Consequences .................................. 33  
   1. Visio Basics ................................................. 33  
   2. Microsoft COM Basics .................................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>Lighting System (LS)</td>
<td>88</td>
</tr>
<tr>
<td>5.</td>
<td>Security Manager (SM)</td>
<td>88</td>
</tr>
<tr>
<td>6.</td>
<td>Database (DB)</td>
<td>89</td>
</tr>
<tr>
<td>D.</td>
<td>Message Sequence Charts</td>
<td>89</td>
</tr>
<tr>
<td>1.</td>
<td>Basic MSCs for locking the automobile</td>
<td>89</td>
</tr>
<tr>
<td>2.</td>
<td>Basic MSCs for unlocking the automobile</td>
<td>90</td>
</tr>
<tr>
<td>3.</td>
<td>High Level MSC for the Central Locking System</td>
<td>91</td>
</tr>
<tr>
<td>E.</td>
<td>Modeling the Central Locking System using M2Code</td>
<td>93</td>
</tr>
<tr>
<td>F.</td>
<td>Interpreting the generated diagrams</td>
<td>95</td>
</tr>
<tr>
<td>1.</td>
<td>KF State Machine</td>
<td>95</td>
</tr>
<tr>
<td>2.</td>
<td>LM State Machine</td>
<td>96</td>
</tr>
<tr>
<td>3.</td>
<td>LS State Machine</td>
<td>97</td>
</tr>
<tr>
<td>4.</td>
<td>SM State Machine</td>
<td>98</td>
</tr>
<tr>
<td>5.</td>
<td>DB State Machine</td>
<td>99</td>
</tr>
<tr>
<td>6.</td>
<td>CONTROL State Machine</td>
<td>99</td>
</tr>
<tr>
<td>7.</td>
<td>CLS System Structure Diagram</td>
<td>101</td>
</tr>
<tr>
<td>G.</td>
<td>Import and Simulation in AutoFocus</td>
<td>102</td>
</tr>
</tbody>
</table>

IX Evaluation | 105 |
| A. M2Code | 105 |
| B. GME | 108 |
| C. AutoFocus | 110 |

X Related Work | 112 |
| A. Introduction | 112 |
| B. Approaches | 112 |
| 1. Labeled Transition System Analyzer - LTSA | 112 |
| 2. Play-Engine | 114 |
| 3. MSC Editor, Simulator and Analyzer - MESA | 116 |
| 4. Minimally Adequate Synthesizer - MAS | 118 |
| C. Comparison | 121 |

XI Conclusion | 122 |
| A. Future Work | 122 |
| 1. Enhancements | 122 |
| 2. Tool Refactoring | 124 |
| 3. Plug-in for Eclipse | 126 |
| 4. Extensions | 127 |
| B. Summary | 128 |

Bibliography | 1
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.1</td>
<td>Sample Use Case and Class Diagram</td>
<td>4</td>
</tr>
<tr>
<td>I.2</td>
<td>Sample Sequence Diagram and BasicMSC</td>
<td>5</td>
</tr>
<tr>
<td>I.3</td>
<td>Sample HMSC and State Machine</td>
<td>6</td>
</tr>
<tr>
<td>I.4</td>
<td>Sample System Structure Diagram</td>
<td>7</td>
</tr>
<tr>
<td>I.5</td>
<td>Toolchain Development at S3E Lab</td>
<td>9</td>
</tr>
<tr>
<td>I.6</td>
<td>BasicMSC Example</td>
<td>11</td>
</tr>
<tr>
<td>I.7</td>
<td>Generated State Machine and SSD Example</td>
<td>12</td>
</tr>
<tr>
<td>II.1</td>
<td>Methodology</td>
<td>15</td>
</tr>
<tr>
<td>II.2</td>
<td>Basic MSC Example</td>
<td>19</td>
</tr>
<tr>
<td>II.3</td>
<td>Basic MSC - Axis, Message, State and REF operator</td>
<td>20</td>
</tr>
<tr>
<td>II.4</td>
<td>Basic MSC - ALT, PAR and LOOP operators</td>
<td>21</td>
</tr>
<tr>
<td>II.5</td>
<td>Basic MSC - JOIN, PREEMPT and TRIGGER operators</td>
<td>22</td>
</tr>
<tr>
<td>II.6</td>
<td>High-Level MSC Example</td>
<td>23</td>
</tr>
<tr>
<td>II.7</td>
<td>High Level MSC - START, END and REF operators</td>
<td>23</td>
</tr>
<tr>
<td>II.8</td>
<td>High Level MSC - Sequence, Preemptive and Triggering connections</td>
<td>24</td>
</tr>
<tr>
<td>II.9</td>
<td>High Level MSC - JOIN and PAR operators</td>
<td>24</td>
</tr>
<tr>
<td>II.10</td>
<td>State Machine Example</td>
<td>25</td>
</tr>
<tr>
<td>III.1</td>
<td>Overview of tool that converts MSCs to State Machines</td>
<td>26</td>
</tr>
<tr>
<td>IV.1</td>
<td>Basic and High Level MSC stencil in Microsoft Visio</td>
<td>34</td>
</tr>
<tr>
<td>IV.2</td>
<td>ShapeSheet for a rectangle in Visio</td>
<td>35</td>
</tr>
<tr>
<td>IV.3</td>
<td>Microsoft COM Basics</td>
<td>36</td>
</tr>
<tr>
<td>IV.4</td>
<td>Microsoft Visio Object Model</td>
<td>37</td>
</tr>
<tr>
<td>V.1</td>
<td>M2Code High Level Design</td>
<td>43</td>
</tr>
<tr>
<td>V.2</td>
<td>Model-View-Controller design pattern</td>
<td>45</td>
</tr>
<tr>
<td>V.3</td>
<td>Composite design pattern</td>
<td>46</td>
</tr>
<tr>
<td>V.4</td>
<td>M2Code Low Level Design</td>
<td>47</td>
</tr>
<tr>
<td>V.5</td>
<td>M2Code Visio Interface Diagram</td>
<td>49</td>
</tr>
<tr>
<td>V.6</td>
<td>M2Code UserInterface Add/Remove Diagram</td>
<td>50</td>
</tr>
<tr>
<td>V.7</td>
<td>M2Code UserInterface Add/Remove/Connect Shapes</td>
<td>51</td>
</tr>
<tr>
<td>V.8</td>
<td>M2Code MSC Domain Model</td>
<td>52</td>
</tr>
<tr>
<td>V.9</td>
<td>M2Code Model Conversion and Generation</td>
<td>53</td>
</tr>
<tr>
<td>V.10</td>
<td>Text file tokenizer</td>
<td>55</td>
</tr>
<tr>
<td>VI.1</td>
<td>AutoFocus project creation and DTD definition</td>
<td>57</td>
</tr>
<tr>
<td>VI.2</td>
<td>AutoFocus Usage Demo - SSD for the system and STD for KF</td>
<td>58</td>
</tr>
<tr>
<td>VI.3</td>
<td>AutoFocus Usage Demo - STD for CONTROL and EET for the system</td>
<td>59</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX.1</td>
<td>User Interface Requirements</td>
<td>106</td>
</tr>
<tr>
<td>IX.2</td>
<td>Translation Requirements</td>
<td>107</td>
</tr>
<tr>
<td>IX.3</td>
<td>Import Export Requirements</td>
<td>108</td>
</tr>
<tr>
<td>X.1</td>
<td>Comparison of different approaches</td>
<td>120</td>
</tr>
</tbody>
</table>
ABSTRACT OF THE THESIS

Building a Tool for Synthesis of Correct Design from Interaction Specifications

by

Praveen N. Moorthy

Master of Science Computer Science
University of California, San Diego, 2006
Professor Ingolf H. Krueger, Chair

Design of large systems is getting very complicated due to the increase in the number of features and the resulting feature interactions. This problem is particularly grave in the area of automotive systems, where 30 - 80 electronic control units interact to deliver thousands of software enabled features to the end user. To avoid surprises in the integration of independently developed components for such systems, it is helpful for developers to be able to precisely specify system interactions in the form of sequence diagrams or message sequence charts since they allow the system behavior to be described visually. Implementing a system that is based on these interactions would be very hard to generate manually due to state space explosion. It is a common practice to start writing code based on the interaction specifications but this can lead to a slew of problems due to unexpected state transitions, deadlocks etc. Moreover, there is no easy way of verifying the validity of the system.

We introduce a tool for the automatic synthesis of system design from interactions specified using message sequence charts. The tool takes user input through a graphical editor and converts this specification into minimized state transition diagrams on a per component level. The tool also has the ability to export the resulting design into other formats for further analysis. Our tool is designed to import and export data in XML formats in order to allow integration of other tools being developed as part of the Service Oriented System Design
methodology being promoted by the S3E Lab at UCSD. We also evaluate the approach with examples from the automotive domain. The result is a comprehensive tool supporting the transition from interaction-based specification to state-machine based analysis and implementation of reactive systems.
Chapter I

Introduction

I.A Problem Statement

Advances in technology and the addition of complex features into embedded systems has resulted in ever increasing complexity of system design. This is especially true in the automotive domain where a large number of embedded controllers are used to control various parts of the automobile. Each of these controllers is designed to perform either one or a set of tasks and in order to complete these tasks, they need to communicate with each other and in some cases share functionality or computational load. The software for such a system becomes very complicated due to the large number of entities involved and their corresponding interactions. Additionally, in order to be able to provide flexibility, it is desirable to design the system in such a way that components from one hardware vendor can be easily replaced by a similar component from another vendor without having to rewrite the software for the entire system. Finally, it is not trivial to verify if the implemented system meets the requirements and to validate if it behaves as desired.
I.B Model Driven Design

This problem of managing complexity in software design has been analyzed in the field of Software Engineering and different approaches have been proposed. Edsger Dijkstra, in his paper on Operating System design [16], describes breaking down the system into modules with well defined interfaces. Barry Boehm, in his paper on Software Development and Enhancement [10] describes early requirement capture and prototyping followed by continuous refinement of both these entities. Booch, Rumbaugh and Jacobson in their efforts to create a Unified Modeling Language [21] [5] [20] describe the approaches of object oriented analysis and design and usage of graphical models to describe the system at various stages of development - from requirement capture to design to final deployment. The efforts in standardizing UML and component oriented design are currently being addressed by the Object Management Group and they are in the process of releasing a formal specification for Model Driven Architecture [4]. Additionally, active research in this area is being carried out at the Vanderbilt University by Janos Sztipanovits [3].

The common theme in the above approaches is to break down the system into smaller, manageable entities - modules/objects/components and inclusion of requirements traceability into system design. Model driven design is not only used in Software Engineering but also in a lot of other fields. In Mechanical Engineering, for example, the first step before starting the fabrication of a mechanical part is to draw diagrams that depict various aspects of the part, to different levels of detail and complexity. Analyzing the part graphically before actually manufacturing it helps in a variety of ways as it allows the designer to show the designs to the customer to make sure that the design meets the requirement and more importantly run simulations on the model to reduce errors in the finished part. The same analogy holds for Software Engineering too.

The model driven design approach in Software Engineering focusses on
using visual abstractions to describe various elements of the system. It also em-
phasizes on carrying out the modeling process in multiple phases. Modeling the
system helps break down some of the complexity of the system into smaller sets
of visual elements that are easy to comprehend. Modeling also allows different
stake-holders (e.g.: Marketing, Systems Engineering, Developers and Testers) in
the system to view the system from different angles.

I.B.1 What are these models?

Model oriented design involves defining various elements in the system, their relationships, their interactions and their roles in abstract terms using dia-
grams. Similar to architectural drawings of buildings, it is not possible to have
one large drawing that describes the entire system. The model of the system is
broken down into different diagrams, each attempting to describe one aspect of
the system. Additionally, these different diagrams may show overlapping pieces of
information, in order to allow the end user of these diagrams the ability to join
the pieces together and visualize the entire system. Based on the different types of
information that is conveyed, there are different types of views as described below.

Use Case Diagrams

Use case diagrams are behavioral diagrams that are used to describe the
different use cases of the system. The initial step in designing a system involves
identifying the various entities and actors in the system and describing their re-
lationships such as dependency, generalization and association. These diagrams
are typically used to model the context or the requirements of the system and
describe the system from a very high level. Figure I.1a shows sample use cases for
an automobile. This diagram shows one actor, the Owner and three use cases -
“Open Driver Door”, “Open Passenger Door” and “Start The Engine”.

Class Diagrams

Object oriented design [33] involves identifying key abstractions in the system and breaking it down into a set of classes or objects. Class diagrams are structural diagrams that describe the various classes in the system, their relationships, the interfaces that they expose, their member functions and member variables. These diagrams are good for describing system decomposition but provide no semantics for describing behavior. Figure I.1b shows a sample class diagram for the parts of an automobile controller. It shows Brake as a generic base class and FrontBrake and RearBrake as two specializations. Additionally, it shows the associations between the BrakeSystem and the Brake - one BrakeSystem can control one or more Brakes; and between the AutomobileController and the BrakeSystem - one AutomobileController controls one BrakeSystem.

![Sample Use Case and Class Diagram](image)

Figure I.1: Sample Use Case and Class Diagram

Sequence Diagrams

Sequence diagrams are behavioral diagrams that describe the interactions in the system under design. They depict objects on a time line and show an ordered exchange of messages between them. These diagrams are typically drawn after the Use Case Diagrams have been drawn and the various use cases defined. They
provide more information than Use Case Diagrams as they describe the various scenarios of system behavior between decomposed system elements (objects) rather than very high level elements (actors). Additionally, these diagrams commonly describe the flow of messages at an object level rather than at a component (group of objects) level. The order in which the messages are exchanged is important as the implementation must exhibit the flow of messages in the same order. Figure I.2a shows a sample sequence diagram describing the series of function calls that occur between different objects in the software for an automobile as a consequence of the user pressing the brake pedal.

![Sample Sequence Diagram](image)

Figure I.2: Sample Sequence Diagram and BasicMSC

**Message Sequence Charts**

Message Sequence Charts (MSCs) are behavioral diagrams that are used to describe distributed, reactive systems by showing timed, asynchronous exchange of messages between the different components. Although Sequence Diagrams and Message Sequence Charts offer similar semantics, they are quite different in their approaches and abstractions. Their differences and possible approaches towards harmonization of the two representations are described in [29]. MSCs can also be used to describe scenarios that must not occur in the system, allowing system
testers to validate the correctness of the system. Since MSCs describe interactions between components, they can be used to capture the requirements of the system.

There are two types of MSCs - Basic and High Level. Basic MSCs define interaction scenarios between components while High Level MSCs describe composition of Basic or other High Level MSCs. Figure I.2b shows a BasicMSC with three components - KF (Key Fob), CONTROL (Controller) and LM (Lock Manager). It describes the scenario of locking an automobile with the exchange of messages between the three components. Figure I.3a shows a sample High Level MSC with three references to other Basic or High Level MSCs - MSC A, MSC B and MSC C. It describes how they are composed together with the scenarios starting with MSC A, upon completion of which, MSC B occurs and upon completion of which MSC C occurs and the scenario ends after that.

![State Machine Diagram](image)

**Figure I.3: Sample HMSC and State Machine**

**State Machines**

State machines are behavioral diagrams that show the various states of the system and the events that cause the transitions between these states. Transitions occur on an initial state, if certain preconditions are met and after their occurrence, the system changes its state to the final state and the post conditions
are applied to this final state. These diagrams can be hierarchical in nature allowing further decomposition of a particular state into smaller state machines. Since these diagrams describe the system to a very fine grained level of detail, they are very close to the actual implementation of the system. Figure I.3b shows a sample state machine for an automobile controller. It describes the different states - *idle*, *locked* and *unlocked* and the transitions *unlock*, *lock* and *exit* between them. The white circle represents the initial state and the black circle represents the final state of the state machine.

**System Structure Diagram**

System Structure Diagrams are structural diagrams that describe the different components in the system and the channels of communication between them. These are very high level diagrams and help the designer/user visualize how many components are there in the system and which components communicate with each other. Figure I.4 shows a sample System Structure Diagram for an automobile controller. It describes three components - CONTROL (Controller), KF (Key Fob) and LM (Lock Manager) and shows the channels of communication between them. Every arrow between two components represents a channel of communication from the component at the tail end of the arrow to the component at the arrowhead end of the arrow.

![Figure I.4: Sample System Structure Diagram](image-url)
I.C Shortfalls in Tool Chains

In order to aid software engineering and development efforts, a lot of tools have been designed and are being used extensively. Unified Modeling Language - UML is one of the widely used languages for modeling software systems. Since UML relies on graphical representation of the system via different diagrams, it is very essential to employ computer based graphical tools that allow the designer to input and analyze the system in greater detail. Tools with varying degrees of complexity are available. Tools such as Microsoft Visio [13] are general purpose graphical editors that provide widgets for representing the various UML diagrams. UML Pad [9], on the other hand is a graphical editor that is specifically designed for UML syntax. These tools are primarily useful for drawing diagrams but further analysis needs to be carried out manually or by using other tools. Rational Rose [32] and ArgoUML [35] are tools that take this one step forward by allowing code generation support. However, this support is limited to skeletal structure, for example generation of c++ header file with function prototypes and blank stubs in the cpp file. What is lacking, is the ability to represent behavioral requirements, specifically interaction requirements in fairly abstract notation and have the tool convert that into concrete terms that can be mapped directly into a programming language. AutoFocus [1] and GME [36] meet some of our requirements but not all and in Chapter IX, we compare our solution with theirs.

I.D Proposed Solution

As part of the ongoing research work being carried out out in the Service Oriented Software and Systems Engineering Laboratory (S3EL) at UCSD, we are designing, implementing and integrating a set of tools that allow the user to describe interaction requirements as MSCs, the tools that analyze and validate these MSCs, convert it into state machines, verify and simulate these generated state machines and finally convert these state machines into executable code for the RT
CORBA middleware. Figure I.5 shows these different tools and their interactions. There overall tool chain comprises of the following tools:

![Diagram showing the toolchain development at S3E Lab](image)

Figure I.5: Toolchain Development at S3E Lab

M2Code

In this thesis, we propose a tool that bridges the gap between MSCs (requirements) and state machines (implementation) - M2Code. We design and implement a tool that supports the full range of MSC operators (composition, join, parallel, alternatives, preemption and trigger) as defined in [23] and implement the algorithms for conversion described therein. The resulting automata are exported as XML for further analysis by other tools.

AutoFocus

AutoFocus [1] is graphical user interface driven modeling tool developed at the Department of Informatics, Technische Universität München. It can be used to describe distributed systems via different views that describe the overall structure of the system, their paths of communication, the behavioral description of the
system or any of the contained components, the interaction of these components via message exchange and internal automata of the component. Additionally, these components can be described in a hierarchical manner allowing further decomposition. The described system can also be run through a simulator allowing visual inspection of the behavior of the system under design. It can also import models from other tools and we utilize this feature to import the models generated by M2Code and simulate them. The details of this are described in Chapter VI.

RTCGen

The RTCGen [7] is a tool designed and implemented in S3EL at UCSD. It takes two pieces of input and generates executable code targeted for the Real Time CORBA middleware. The first piece of input is an XML file consisting of component level automaton, system structure and QoS constraints for the system under design and the second piece of input is a set of templates that control the content and behavior of the code being generated. The generated models from M2Code serve as the first piece of information and the details of these are described in Chapter VI.

I.E Example

In order to better understand our proposed solution and the usage of M2Code, we describe a sample scenario from the automotive domain. We describe a BasicMSC and the generated state machine and system structure diagram. This section serves as a glimpse to the various aspects of the proposed solution. The subsequent chapters dive into more details in various aspects such as the notation, the methodology, algorithm etc.

The BasicMSC in Figure I.6 shows four components - KF (Key Fob), CONTROL (Controller), LM (Lock Manager) and LS (Lighting System). This BasicMSC describes the interactions for unlocking an automobile by the exchange
of the different messages between the various components in the software of the controller. The hexagonal shapes on the components are state markers that are added by the designer to describe perceived component state at that given point in time. For example, the CONTROL component starts in the LCKD state and at the end of this scenario, it moves to the UNLD state. Similarly, the KF component starts in the INITIAL state and returns back to this state after receiving the ok message from the CONTROL component.

Using our tool, we would like to convert this BasicMSC into a set of state machines for each component and one overall system structure diagram. Figure I.7a shows the converted state machine for the KF component. It shows _js0_ as both the initial and final state and sending the unlk message to the CONTROL component causes a transition to state _js1_ and the receipt of the ok message from the CONTROL component causes a transition back to the _js0_ state.
Figure I.7b shows the overall system structure diagram based on the components and their channels of communication described in the BasicMSC shown in Figure I.6a. For example, this diagram shows that there are two channels of communication between the KF and the CONTROL components - CK4 from the CONTROL to the KF component and KC0 from the KF to the CONTROL component.

Figure I.7: Generated State Machine and SSD Example

I.F Contributions and Outline

The path from requirements to design to code is getting very complicated as systems have complex requirements leading to a large number of expected and unexpected feature interactions and state space explosion. In this thesis, we propose a solution to alleviate some of this complexity by identifying the requirements for a tool that generates implementation views from interaction specifications. We also describe the architecture for such a tool and describe solutions to problems that can occur during the design and integration of such tools. We also analyze other tools and approaches that attempt to address same/similar problems and evaluate our approach to theirs.

This thesis describes the tool M2Code that was designed to address the problem statement described in Section I.A. We explain the methodology behind
our approach in Chapter II. The requirements of a tool that would implement the
proposed solution are described in Chapter III. Using these requirements as the
starting point, we describe the motivation and rationale behind our approach in
Chapter IV. In Chapter V we describe the architecture and design of the tool in
detail. Rather than implementing features such as simulation and code generation,
which are already available in other well known tools, we leverage these features
from other tools by providing interfaces to these tools, the details of which are
described in Chapter VI. The algorithms involved in conversion of MSCs to state
machines is described in Chapter VII. In order to understand the problem and the
solution better, we take a problem from the automotive domain - Central Locking
System for an automobile and walk through the steps of using M2Code for this
problem domain in Chapter VIII. There are other tools that attempt to address
some of the problems described in Section I.A. We evaluate our tool against the
tool requirements with two other tools - AutoFocus and GME in Chapter IX. We
also analyze related research in the field of automated generation of implementation
from requirements in Chapter X and compare and contrast our solution with theirs.
Finally, in Chapter XI we provide insights into how the tool could be enhanced in
the future and what other extensions could be added to it in order to help achieve
the goals of the research work being carried out at S3EL.
Chapter II

Methodology and Notation

II.A Introduction

In Chapter I, we described the various approaches in model driven design and how in our proposed solution we chose MSCs to describe interaction specifications and use a tool to convert them into state machines that describe fine-grained implementation level detail. In this chapter, we describe our methodology and provide an intuition to the conversion process from MSCs to state machines. We also provide an introduction to the notation and semantics used in MSCs and state machines.

II.B Methodology

In this section, we describe the methodology for the design of distributed, reactive systems such as the embedded controller of an automobile. These types of systems are characterized by the presence of multiple processors or microcontrollers each performing a well defined set of functions and communicating with each other asynchronously by passing messages. Figure II.1 provides an insight into the methodology that we propose in this thesis. The rectangles in the diagram represent different stages in the design of the system and the arrows point to the next possible stage of the design. This approach adopts aspects from different
software development processes such as the Spiral Model [10] and the iterative and waterfall models described in [18]. It relies heavily on models to describe the system visually, in order to make it easier to understand and comprehend. In addition to this, it also supports constant refinement of the output of each stage using feedback from any of the stages that follow it. In Chapter VIII, we apply this methodology to a case study for the central locking system of an automobile. The different stages of the methodology are described in detail in the following sections.

![Figure II.1: Methodology](image)

### II.B.1 Requirements

This is the first stage of the process and involves identifying the different requirements of the system. These are generally very high level and primarily consist of an itemized list of what the system must do. Usually, these requirements are provided by the Marketing group within the company designing the product or an external customer. These stakeholders typically are not very technical and hence the requirements are described in very simple, layman terms. These can either be described textually in a document or graphically using the Use Case Diagrams described in UML [5].

### II.B.2 Component Identification

This next stage in our methodology involves categorizing the different requirements identified in the Requirements stage and identifying key abstractions or the components in the system. The concepts of modularizing and layering described in [16] and [18] are applied in this step. Additionally, considerations can be
made based on the similarity or locality that is desired from a set of requirements. For example, all the requirements that relate to handling, operating and control of all the brakes in the automobile can be grouped together into one category and handled by a component - the BrakeManager. At the end of this stage, the designer(s) must be able to identify the major components of the system based on the considerations mentioned above. For the automobile example, based on the requirements, we could envision the following components - BrakeManager, LockManager, Controller, AudioSystem, Navigator, SteeringSystem, SecuritySystem, KeyFob, LightingSystem etc.

II.B.3 Interaction Specification

Once the components are identified as mentioned in Section II.B.2, we need to convert the high level requirements described in II.B.1 into detailed scenarios. This process involves taking each requirement and finding the component that was designed to implement that requirement and representing it in the form of a scenario. There are different types of graphical representations that are used to represent scenarios. The commonly used ones are Sequence Diagrams, defined as part of the UML specification [5] and Message Sequence Charts, defined as part of the MSC-96 specification [19]. Our methodology involves usage of the MSC standard, wherein the different components interact asynchronously by exchanging a sequence of messages. The specified requirements may be translated into any number of scenarios but every requirement must occur in at least one MSC.

II.B.4 System Structure and State Machines

Once the interaction scenarios are described in the form of MSCs as described in II.B.3, they need to be converted into an intermediate model that can be used for verification by model checkers and for conversion into implementation by code generators. We propose the usage of System Structure Diagrams (SSDs), to represent the structure of system under design and state machines to represent
implementation level details of a component level.

This conversion from interaction scenarios into SSDs involves iterating through all the components in all the MSCs and iterating through all the messages sent or received by them. This information can be used to interpret the channels of communication between the components, their direction and the messages that they carry. For example, if two components A and B send and receive messages with each other, we can interpret the existence of two channels between them - channel AB that carries messages from A to B and channel BA that carries messages from B to A.

The conversion of MSCs to state machines is fairly involved and there are a lot of approaches available. We have taken the approach described in [23] and we describe implementation of these algorithms in Chapter VII. We have also investigated other approaches and we describe these in Chapter X. The basic theme in all of these approaches is that they consider every exchange of messages (either sending or receiving) between components as a state change. The actual message that was exchanged is considered as the transition that causes this change in state. There are two approaches for analyzing this state change - state changes at a component level or system-wide state change. We take the former approach as the system-wide state can be very large for complex systems. Moreover, the coupling between components in a distributed system is very low, making state only at a component level more relevant. Apart from the these states that are added during the conversion process, the designer is also encouraged to add state markers on various points of the Basic MSCs to aid the the conversion process. The final step of the conversion process involves combining the state machines generated from every MSC for every component based on the compositional operators defined in the Basic and High-Level MSCs.

This stage of our process consists of complex conversion process, hence we propose the usage of a tool rather than carrying out this process manually. The design of such a tool is the motivation for this thesis.
II.B.5 Implementation

This is the final stage of our process. In this stage, the generated SSDs and state machines are converted into executable code. This is also an involved process and we again propose using a tool for this. The conversion process that we have proposed as part of the research at S3EL, is described in detail in [7] and hence is not discussed in too much detail in this thesis.

II.C Notation

In this section we describe the notation for the different types of diagrams that are used in this thesis. Note that this does not serve as a replacement for formal semantics, nor does it attempt to describe the detailed reasoning behind them. The intention here is to introduce the various symbols in order to help the reader better understand our proposed solution. Please refer to [23] for formal semantics, detailed description and a comparison of the different notations and their pros and cons. Also note that a some of this notation is in the process of being included into the UML2 [5] specification.

II.C.1 Message Sequence Charts

Message Sequence Charts (MSCs) are diagrams that are used to describe interaction scenarios, typically between distributed components in the form of asynchronous flow of messages. There are different dialects of MSCs and the one that is widely used is the MSC-96 specification [19] which, we have also adopted in this thesis. There are two types of MSCs that we refer to - Basic MSCs (BMSCs) and High-Level MSCs (HMSCs). The different symbols and operators used in both of these are described below.
Basic MSCs

These are MSCs that are used to describe asynchronous flow of messages between components. Figure II.2 shows a Basic MSC (BMSC) consisting of two components for the software for an automobile - KF (Key Fob) and CONTROL (Controller).

This BMSC describes the interaction scenario consisting of the parallel composition of two messages lck and ok with another MSC - UNLK-2. It also shows how the state of CONTROL changes from UNLD to LCKD as a consequence of receiving the lck message. This example consists of only a few of the operators that can be present in a BMSC. The following section describes all the operators of BMSCs that are used in this thesis and their meaning in brief.

In BMSCs, components are described by using the Axis graphical element as shown in Figure II.3a. The name of the component is placed above the upper
Components may change their internal state based on the exchange of messages with other components. Users can describe a components’ perceived state by placing state markers along the time line of the component using the shape shown in Figure II.3c. In the time line of a component in an MSC, a component may exchange a set of messages that is already described as an interaction scenario in another MSC. Rather than repeating the same set of messages in both the MSCs, the user can place a reference to the second MSC by using the REF operator shown in Figure II.3d. After the completion of the interactions described in the referred MSC, the scenario proceeds with the interactions specified on the time line following the REF operator.

In addition to describing simple interaction scenarios by the exchange of messages, BMSCs also support other logical operations that can be applied to
groups of messages, states and references. Based on the functionality supported by these operators, they can operate on one ore more groups of interactions, referred to as compartments. Alternative flow of messages is described using the ALT operator shown in Figure II.4a. Based on the value of a state variable, the flow of interactions would be either those present in the first or the second compartment. Parallel flow of interactions is described using the PAR operator shown in Figure II.4b. Here, the flow of interactions would consist of interactions from both the compartments causing an interleaving of messages to occur, however maintaining the relative order of interactions within the compartment itself.

Repetitively occurring interactions can be grouped together using the LOOP operator shown in Figure II.4c. This operator contains only one compartment and all the interactions contained within this compartment will be executed repeatedly until the loop invariant evaluates to false, at which point the interactions continues with the interactions following the LOOP operator.

![Figure II.4: Basic MSC - ALT, PAR and LOOP operators](image)

The JOIN operator can be used to combine a set of interactions based on one or more common messages between the two interactions, using the operator shown in Figure II.5a. The messages in the compartments are combined by synchronizing the interactions on the common set of messages between the compartments and interleaving on the remainder of the messages.

The PREEMPT operator is used to describe interactions that need to occur in case certain exceptional messages occur using the operator shown in Figure
II.5b. The messages that cause the preemption to occur are shown in the box in the upper right corner of the PREEMPT operator. The interactions that need to be preempted are shown in the first compartment and the interactions that need to occur once the preemptive message is encountered are shown in the second compartment.

Interaction specifications that cause the eventual execution of another set of interactions can be described using the TRIGGER operator shown in Figure II.5c. The set of interactions that cause the trigger are shown in the first compartment and the interactions that eventually occur after the occurrence of the trigger are shown in the second compartment.

![JOIN, PREEMPT, TRIGGER operators](a) (b) (c)

Figure II.5: Basic MSC - JOIN, PREEMPT and TRIGGER operators

High-Level MSCs

High-Level MSCs (HMSCs) are used to compose two or more BMSCs or HMSCs into one overall interaction specification. Figure II.6 shows a sample HMSC describing the composition of two other MSCs - LCK-1 and UNLK-1. It also indicates that the interactions start with those described in LCK-1, followed by those described in UNLK-1 and after the completion of all the interactions in it, the overall interaction specification comes to an end. This example only shows some of the operators that can be used in an HMSC. The following section describes the HMSC operators that are used in this thesis with a brief description of their usage and meaning.
In HMSCs, the different MSCs are referenced using the REF operator shown in Figure II.7c. The start and end of the MSC composition is described using the START and END operators respectively as shown in Figures II.7a and II.7b.

![Figure II.7: High Level MSC - START, END and REF operators](a) (b) (c)

The interpretation of the composition of the MSCs is carried out by means of different types of connectors. The sequencing connector shown in Figure II.8a is used to connect two MSCs as a sequence. After completion of the interactions specified in the MSC connected to the tail end of the connector, the interactions specified at the arrowhead of the connector are executed.

Preemptive behavior is described by using the preemptive connector shown
in Figure II.8b. The message causing the preemption is labeled on top of the message and the MSC being preempted is connected to the tail end of the connector and the MSC that describes the interactions that need to occur as a consequence of the preemption is connected to the arrowhead of the connector.

MSCs that can trigger the execution of other MSCs are described using the triggering connector shown in Figure II.8c. The MSC that causes the trigger to occur is connected to the tail end of the connector and the MSC that describes the interactions after the occurrence of the trigger is connected to the arrowhead of the connector.

![Figure II.8: High Level MSC - Sequence, Preemptive and Triggering connections](image)

The operators for describing the join and parallel composition of MSCs are shown in Figures II.9a and II.9b. These are similar to the JOIN and PAR operators described in BMSCs except that their compartments can only contain REF operators, with references to other BMSCs or HMSCs.

![Figure II.9: High Level MSC - JOIN and PAR operators](image)
II.C.2 State Machines

State machines are behavioral diagrams that describe a system in the form of discrete states and transitions that occur between these states. There are slight variations in the notation and semantics of state machines. In this thesis, we use the notation that is described in [23]. A sample state machine is shown in Figure II.10. This state machines describes the set of transitions of a component B. The two circles represent the initial (white circle) and final (black circle) states and the ellipses represent the different states that the component may exhibit - s0, s1 and s2. The arrows between these states represent the transitions that cause the state of the component to change. The state on the tail end of the arrow represents the state before the transition and the state at the arrowhead end represents the state after the transition. The actual message and the channel on which the message is received is described by the label next to the transition. For example, the transition from the initial state to state s0 is labeled as AB!m0. This indicates that the message $m0$ is received on channel AB. A prefix on the message is used to indicate the direction of the message - exclamation mark indicates that the message is received and the question mark indicates that the message is sent. Additionally, state machines provide semantics for their hierarchical decomposition allowing states to be composed of state machines themselves.
Chapter III

Tool Requirements

In Chapter I we described the challenges involved in the design, analysis and implementation of complex reactive statements and explained the need for a tool that would help alleviate some of the problems. Based on these we enumerate the overall requirements of such a tool.

III.A General Overview

![Figure III.1: Overview of tool that converts MSCs to State Machines](image)

In order to visually describe the system, it is essential to provide a tool that has a graphical user interface so that the user can diagrammatically input information that corresponds to the various views of the system. Using these initial views that describe the requirements of the systems, the tool should be able to automatically generate views that show and describe the implementation strategy for
that system. As indicated in Section I.D, in our model oriented design approach, we have chosen to use MSCs (Basic and High-Level) as views for describing system requirements and state machines and System Structure Diagrams (SSD) as automatically generated output that describes the implementation. So, the user should be able to use the tool to draw Basic and High-Level MSCs and the tool should be able to automatically generate state machines and SSDs. Additionally, in order to be able to easily extend the functionality of the tool, it must also be able to export the generated views into different formats. Based on this, we break down the requirements into three categories - requirements that are specific to the user interface, requirements that pertain to the translation process from MSCs to state machines and requirements that pertain to the importing and exporting of the model into different formats.

### III.B User Interface Requirements

1. The tool must provide a graphical canvas interface for the creation of Basic and High-Level MSCs.

2. The canvas interface must allow the user to add graphical widgets and their interconnections.

3. The tool must provide graphical widgets in Basic MSCs for describing:
   
   (a) a component and its life cycle on a time-line.
   (b) the state of a component at a certain point in time on its time-line.
   (c) sending/receiving message between components.
   (d) references to other Basic MSCs.
   (e) the parallel composition of messages or references to other Basic MSCs.
   (f) the join of groups of messages or references to other Basic MSCs.
   (g) alternative message exchanges between components.
(h) looping over a group of messages or references to other Basic MSCs.
(i) the preemption of a group of messages or references to other Basic MSCs with a group of messages or references to other Basic MSC on the receipt of one or more messages.
(j) the triggering of a group of messages or references to other Basic MSCs due to the occurrence of a group of messages or references to other Basic MSCs.

4. The tool must provide graphical widgets in High-Level MSCs for describing:
   (a) start of the MSC.
   (b) end of the MSC.
   (c) references to other Basic and High-Level MSCs.
   (d) the parallel composition of other Basic or High-Level MSCs.
   (e) the join of other Basic or High-Level MSCs.
   (f) the interconnections in order to describe the type of composition of the elements as either sequential, preemptive or triggering.

5. The tool must provide an interface to select the components that need to be converted from a list of all the components in the current model.

6. The tool must provide an interface to list all the Basic and High-Level MSCs that a particular component appears on and allow the user to select from this list the MSCs that need to be included in the conversion process.

7. The tool must provide an interface to list all the state markers that the user as added to a particular component and allow the user to select from this list the initial and final states for the conversion process.

8. The tool must be able to construct a visual representation of the generated state machine using graphical layout information.
9. The tool must be able to construct a visual representation of the generated SSD using graphical layout information.

10. The tool must provide an interface to allow the user to store the Basic and High-Level MSCs on persistent storage.

11. The tool must provide an interface allow the user to load Basic and High-Level MSCs from persistent storage.

III.C Translation Requirements

1. The tool must be able to convert Basic and High-level MSCs into state machines on a per component level given the initial and final states for that component, using the algorithms described in [23].

2. The tool must be able to minimize the generated state machine.

3. The tool must be able to generate graphical layout information for the generated state machine.

4. The tool must be able to deduce the channels of communication between different components using the message flow described in the selected MSCs and generate an SSD that represents the selected components and the channels of communication between them.

5. The tool must be able to generate graphical layout information for the generated SSD.

III.D Import Export Requirements

1. The tool must be able to store the Basic and High-Level MSCs in a format that can be used by the tool to load these models at a later point in time or as required.
2. The tool must be able to export the generated state machines, SSDs and the associated layout information into the Query Markup Language (QML) format supported by AutoFocus.

3. The tool must be able to export the generated state machines and SSDs into the Extensible Markup Language (XML) format supported by RTCGen.

4. The tool must be able to export the Basic and High-Level MSCs, the state machines and the SSDs into formats that can be imported by other graphical editors such as Microsoft Word, Microsoft PowerPoint.
Chapter IV

Design Rationale, Decisions and Consequences

IV.A Design Rationale

In Chapter III we described the various requirements for designing a tool that would meet our objectives. Before getting into the actual architecture and design of the tool, we would like to provide some background information about our approach and explain the rationale behind it.

One of the important issues during any interactive tool design is user interface. The adoption of any tool into a software engineering process largely depends on its ease of use. Since we needed the ability to enter MSCs as an input to our tool, we could take one of the following approaches:

- Design our own editor from scratch: This would have provided us with a lot of flexibility in terms of how and what to present to the user. However, the development of a full featured graphical editor could be very time consuming and could potentially be a research project in itself!

- Modify an open source editor: This would provide us with some amount of flexibility since we could modify the source, if required. However, most of the open source graphical editors are used for image manipulation and are
hence not very well suited for our requirements. Additionally, this approach would tie our implementation to that tool making it harder to migrate to other tools in the future, if required.

- Plug-in into full featured editor: This approach would give us lesser flexibility on the user interface as the editor designers have their own set of requirements to implement. However, a stable tool, with good interface for plug-ins would provide us with a stable platform for initial tool development and would also provide sufficient amount of decoupling with the editor. This would also allow us to concentrate more on the conversion algorithms rather than on the graphical interface issues.

Based on the above consideration, we decided to go with the third approach as it allowed us to concentrate on the implementation of the algorithms without having to be overly concerned with the design of the graphical editor. Taking this approach, however, meant that we needed to find a stable tool with good support for customization. We also realized that we could extend this approach of reusing existing stable tools to other areas such as state machine minimization and diagram layout generation.

With regards to the user interface, we found Microsoft Visio [13] to be a widely used graphical editor and decided to use it as the front end for the tool. Additionally, it exposes rich interfaces and provides user programmability via COM interfaces [14]. In the Sections IV.2, IV.B.2 and IV.B.3, we describe Visio, its usage, its features and its support for external control in detail and explain how these properties allow us to fulfil the user interface requirements described in Chapter III.

For state machine minimization, we decided to use a readily available finite automata library FA and for the diagram layout generation, we decided to use DOT, a layout tool provided as part of the Graphviz project from AT&T [28]. Both of these are described in Sections IV.B.4 and IV.B.5 respectively.
IV.B  Decisions and Consequences

In this section we describe the different tools that we decided to use, how these decisions help us fulfil the requirements and consequences of these decisions.

IV.B.1  Visio Basics

Visio is a very powerful and widely used graphical editor. It provides a canvas like user interface for users to draw different shapes and objects. This allows us to fulfil requirement III.B1. In addition to the graphical canvas interface, it also provides standard menus for creating new diagrams, opening existing diagrams and saving modified diagrams. We utilize these features to meet the user interface requirements III.B10, III.B11 and import/export requirement III.D1. It categorizes diagrams into three types - drawings, templates and stencils. They are all internally stored by Visio in the same format but they serve different functions. In order to ease drawing, Visio provides some pre-drawn graphical elements that can be logically grouped together into a diagram, which is referred to as a stencil. Users can create diagrams by dragging and dropping these elements from the stencil into the diagram and connecting them using the different types of connectors made available by the tool. Additionally, if diagrams need to be created using a common set of stencils and have common diagram parameters such as size, orientation, macros etc., templates can be defined that abstract this behavior into a diagram.

Users can create a diagram based on a template, in which case, a new diagram is created with all the behaviors defined in the template. Visio defines a large set of stencils with elements for different types of drawings and classes of templates for different activities such as software engineering, mechanical engineering, flowcharts etc. Users are also free to define their own stencils and templates. We decided to design our tool to use this aspect of Visio to define a stencil with widgets for Basic and High Level MSCs. These widgets comprise of the ones mentioned in
requirements III.B3 and III.B4. We also defined a template with references to the above stencil to make it easy to use the tool. The stencil for our tool is shown in Figure IV.1.

Figure IV.1: Basic and High Level MSC stencil in Microsoft Visio

In order to modify the characteristics of any shape in a diagram, Visio introduces the concept of a ShapeSheet, which is a matrix of name-value pairs describing various characteristics of the shape. The ShapeSheet for a rectangle in Visio is shown in Figure IV.2, which shows the various attributes of the rectangle such as width, height, orientation, geometry and coordinates of the shape on the diagram. The various attributes in the ShapeSheet are referred to as cells in Visio.

Since Visio is an application built by Microsoft, it supports the functionality to copy/paste to/from the Windows clipboard. This feature provides us with
the capability to draw MSCs in Visio and easily cut ‘n’ paste them into other Microsoft tools such as Microsoft Word and Microsoft PowerPoint, thereby allowing us to fulfil import export requirement III.D4.

**IV.B.2 Microsoft COM Basics**

Microsoft COM [12] is an object framework provided as part of the Windows operating system. This framework attempts to provide a standardized means of deploying components (or applications) and providing the ability for clients to query and invoke methods on the component (or application) at runtime. COM
takes the approach of defining pure virtual interfaces, thereby separating the interface from the implementation. It also provides support for resource management by exposing APIs for reference counting.

The class diagram in Figure IV.3 is an overly simplified representation of two COM components ComponentA and ComponentB. IUnknown is the base interface for all COM components. The COMFactory class represents the framework within Windows that is used to manage the life-cycle of all COM components, i.e. it is not an actual class within the COM framework. It is just used in this diagram as an abstraction to make it easier to understand the core concept. A client intending to use ComponentA would use the COMFactory to create an instance of ComponentA. This client can now invoke the functions exposed by IComponentA on the object instance received from the factory. This concept allows not only for the deployment and usage of componentized libraries but also for programmatical control of user interface driven tools by other tools. Tools such as Visio allow interaction from the user by providing a user interface in the form of menu’s, dialog boxes, drag ‘n’ drop interfaces etc. In addition to this, it also exports interfaces via COM, thereby allowing programmatic control of its functionality by invoking
these COM interfaces. For a detailed explanation of COM, implementation and deployment issues please refer to [11].

IV.B.3 Visio Internal Architecture

Visio has an internal model that is very object oriented. A small subset of the COM interfaces and their relationships is shown in the class diagram in Figure IV.4. Note that not all the methods/attributes are mentioned in the diagram.

Figure IV.4: Microsoft Visio Object Model

IVApplication represents an instance of the Visio application. This is the first object that must be obtained in order to gain automation access to Visio. This object can be used to get references to all the documents are currently open in that
instance of Visio. It can also be used to change the look ‘n’ feel of the application by changing some of the custom menus and their entries. We use this feature in Visio to draw custom menus, forms etc in order to fulfil the requirements III.B5, III.B6 and III.B7. The IVDocument class represents a document that the user can load, store and modify in Visio. This class maintains a list of the different pages that are present in the document and the different stencils that are being referenced. The IVPage class represents a particular page in the document. This class provides the functionality to draw on that page and methods to iterate through all the shapes and their connections.

The IVMaster class represents the stencil object that is the master copy of the shape that is actually drawn on the page. Refer to Section IV.B.1 for more details regarding stencils. The IVShape class represents shapes that can be drawn on pages. The IVShape class can contain a list of other IVShape’s since shapes can be composed of other sub-shapes. The properties of a shape can be changed by modifying the attributes in its ShapeSheet using the IVCell interface. In order to obtain the various connections between shapes, the IVConnect interface can be used. IVShape’s contain a list of connections for the various shapes that they are connected to. IVPage’s, on the other hand contain a list of all the connections between all the shapes on that page. For more details on using automation in Visio please refer to [37].

These COM interfaces described above that are exposed by Visio allow us to programmatically query each diagram, find out the different widgets in the Basic and High-Level MSC and build an internal model that represents all the MSCs, the different components and their relationships. Using this information and the conversion algorithms described in [23], we can convert Basic and High Level MSCs into state machines, thereby fulfilling the translation requirement III.C1. Additionally, it can also be used to iterate through all the components in all the MSCs and based on their connectivity, build a list of channels that exist between these components. Using this information, we can construct a System
IV.B.4 State Machine Minimization using FA

In Section IV.B.3, we described how we could utilize the COM interfaces in Visio to build an internal model of the MSCs and convert these to state machines. These generated state machines can contain ε transitions added during the conversion process that need to be eliminated. We decided to use a Java-based finite automata library - FA, implemented by Dr. Ingolf Krueger to perform this minimization and fulfil the translational requirement III.C2. This library expects as input a set of transitions, a set of initial states and a set of final states and returns the minimized deterministic finite automaton for that set of state transitions.

IV.B.5 Diagram Layouts using DOT

After the minimized state machine and system structure diagrams are generated by our tool, we need to also generate graphical layout information for these diagrams so that they can be displayed back to the user as mentioned in translation requirement III.C5 and user interface requirements III.B8 and III.B9 respectively.

In order to accomplish these requirements, we decided to utilize a widely used graphical layout generator utility - DOT that is part of the Graphviz Graph Visualization Software developed by AT&T Research [28]. It takes a directed graph as an input and calculates layout information for the graph and outputs this information in a variety of formats such as gif, jpeg, png, text etc. DOT is invoked in the command line mode as follows:

dot -Tplain input.txt -o output.dot

The -Tplain option is used to indicate that the output format is plain text. In order to explain its usage, a sample input file input.txt is shown below:
digraph component {
    s0 -> s1 [label = "?m1"];  
    s1 -> s2 [label = "!m2"];  
    s0 -> s4 [label = "!m4"];  
    s4 -> s2 [label = "?m5"];  
    s2 -> s3 [label = "?m6"];  
}

The input file describes a state machine in a very simple and intuitive form. The first line is used to give a name to the graph and the subsequent lines enumerate all the nodes and the edges between them. The output provided by DOT for this input file is as shown below.

```plaintext
graph 1.000 1.986 4.556
node s0 0.889 4.194 0.750 0.500 s0 solid ellipse black lightgrey
node s1 0.375 2.917 0.750 0.500 s1 solid ellipse black lightgrey
node s2 0.889 1.639 0.750 0.500 s2 solid ellipse black lightgrey
node s4 1.403 2.917 0.750 0.500 s4 solid ellipse black lightgrey
node s3 0.889 0.361 0.750 0.500 s3 solid ellipse black lightgrey
edge s0 s1 7 0.722 3.972 0.667 3.889 0.611 3.792 0.569 3.694 0.486 3.528 0.444 3.319 0.417 3.167 "?m1" 0.847 3.556 solid black
edge s0 s4 7 1.028 3.958 1.069 3.875 1.125 3.778 1.167 3.694 1.236 3.528 1.306 3.319 1.347 3.167 "!m4" 1.500 3.556 solid black
edge s1 s2 7 0.417 2.667 0.444 2.514 0.486 2.306 0.569 2.139 0.611 2.042 0.667 1.944 0.722 1.861 "!m2" 0.833 2.278 solid black
edge s2 s3 4 0.889 1.389 0.889 1.167 0.889 0.833 0.889 0.611 "?m6" 1.181 1.000 solid black
edge s4 s2 7 1.333 2.667 1.292 2.514 1.222 2.306 1.153 2.139 1.111 2.056 1.069 1.958 1.014 1.875 "?m5" 1.528 2.278 solid black
stop
```

The output format provides information in the form of nodes and edges in an easily tokenizable format. The first line is of the form:

```plaintext
graph scalefactor width height
```

This line describes the scaling that needs to be applied when rendering the diagram in any other graphical editor and the width and the height of the entire drawing. The lower-left corner of the drawing is considered as the origin. All the coordinate pairs that are provided in the next set of lines are based on this origin. The next group of lines lists the nodes and their layout information as follows:
node name x y xsise ysize label style shape color fillcolor

The *name* represents the name of the node provided in the input file. The *x* and *y* values give the coordinates of the center of the node; the *xsize* and *ysize* give its width and the height. The remaining parameters are extra values that are copied from the users input, if provided, otherwise contain default values. The next group of lines lists the layout information for the edges:

edge tail head n x1 y1 x2 y2 : : : xn yn [ label lx ly ] style color

The *tail* and *head* are cross-reference to the two nodes that are connected by this edge. The edges are described as a set of B-spline control points and *n* is the number of coordinate pairs (x1, y1) to (xn, yn) that follow this value. If the edge is labeled (using the *label* keyword in the input file), then the label text and the coordinates where this text should be placed are listed next. The style and color are copied from the user input.
Chapter V

M2Code Architecture

V.A Introduction

In Chapter III, we described the requirements of a tool that would address the problems described in Section I.A. In Chapter IV, we introduced the design and provided a rationale behind our decisions and its related consequences. In this chapter, we describe the architecture of the tool by describing the high level design and low level design and explain the different modules and their key abstractions.

V.B High Level Design

The tool framework can be broken down into functional blocks as shown in Figure V.1. The different function blocks are explained below:

- **UserInterface**: This functional block is involved in all the interactions with the user. It is responsible for providing the graphical canvas interface for the user to draw the Basic and High Level MSCs. Additionally, it will take input from the user in the form of menus and forms for the conversion process and provide the functionality for generated diagrams to be drawn by the tool and presented to the user. This module communicates with the ModelConverter by allowing it to synchronously request user interface information and also
provide asynchronous notification of changes in the user interface as and when they occur.

- **ModelConverter**: This is the core functional block of the tool. It is the responsibility of this block to gather all the diagrams and other forms of input from the UserInterface, convert the model into a state machine, pass this state machine through the minimizer, pass all the generated diagrams through a layout generator and invoke the UserInterface to present these generated diagrams to the user. Additionally, it will invoke the ModelExporter to export the converted models into different formats so that it can be imported into other tools.

- **StateMachineMinimizer**: The state machines generated by the ModelConverter contain \( \epsilon \) transitions, added as a consequence of the conversion process. This functional block is used to remove these and minimize it into a deterministic finite automaton.

- **LayoutGenerator**: The state machines and system structure diagrams generated by the ModelConverter need to be presented to the user via the UserInterface. In order to accomplish this, the ModelGenerator invokes this func-
tional block to calculate graphical layout information consisting of relative coordinate positions for the different graphical elements in the generated diagrams.

- ModelExporter: The model generated by the tool framework by converting the MSCs into state machines and an SSD need to be exported into different file formats that can be understood by different tools. These tools will then be able to import these generated models for further analysis. The Model-Converter invokes this module to export the converted model into different file formats.

V.C Design Patterns

Design patterns are proven solutions to recurring problems in software. These problems are identified and characterized into a pattern and a proposed design is suggested, primarily using object oriented design but independent of any programming language. Design patterns were first introduced in a research paper and eventually converted into a recipe style book [17]. In the process of designing our tool framework, we have utilized some of these patterns and we describe them in brief below. In the sections that follow, we will utilize one or more of these patterns and refer them with the pattern name.

V.C.1 Model View Controller

A commonly used design methodology for user interfaces is the Model View Controller (MVC) design pattern [34]. In this design approach, the internal data representation is referred to as the Model, the visual representation is referred to as the View and the entity that processes the user input, updates the internal and external representation is referred to as the Controller. This allows decoupling between the internal and external representations. The class diagram in Figure V.2 describes a simple Model-View-Controller design. In this design, there is one
controller that maintains a collection of views and the internal model. The Controller, creates the different Model elements and View elements as required. Views register for event notifications with Models that they display information about. The Controller receives input from the user and modifies the Model appropriately. This change in the Model causes it to send events indicating the change, which causes the View(s) to update the information being presented to the user.

![Diagram of Model-View-Controller design pattern](image)

Figure V.2: Model-View-Controller design pattern

**V.C.2 Composite**

There are situations where we need to describe a class of elements that can themselves be composed of parts of the same type. For example, we can define a Shape class to represent the different shapes that can be drawn. Simple shapes can be created using this basic type. However, rather than defining a new base class for complex shapes, using this pattern, we can define a new class ComplexShape that derives the basic behavior from the Shape class, and additionally provides a container to hold references to other Shape objects. This allows all shapes simple or complex to be accesses using the basic set of interfaces provided by the Shape class. Figure V.3 shows a sample implementation of a composite design pattern for different types of shapes. The interfaces `draw` and `getArea` are common to both
simple and complex shapes.

Figure V.3: Composite design pattern

V.C.3 Singleton

This is a simple design pattern that is used to restrict the number of instances of a class. There are a lot of circumstances when the number of instances needs to be limited, for example the root model element of a model tree. This pattern utilizes the behavior of static member functions, which allow class specific behavior to be common to all the instances of the class. The key however, is that the static attribute of the class is of the same type as the class itself. This allows, the static instance accessor function to first check if an instance already exists and return it if it exists, if not create one before returning it. The following code fragment shows the sample implementation of a singleton class in c++.

class RootModel {
    public:
        static RootModel* getInstance()
        {
            if (poInstance == null)
            {
                poInstance = new RootModel();
            }

            return poInstance;
        }
}
V.D Low Level Design

The Figure V.4 shows the low-level design of our tool. Since the UserInterface and ModelConverter blocks are the key elements of the tool, these blocks are explained in more detail in the diagram and in the sections that follow. The UserInterface module is shown as two parts - UserInterface(Visio) and UserInterface(M2Code). This is due to the decision decisions we made early on and the rationale for this is described in Section IV.A, where we mentioned that we would use Visio for user interactions. However, since we don’t have the source code for it, we have to attach our tool using its established interfaces via COM as explained in Section IV.B.3. Due to this, we need to have a part of the UserInterface module within our tool and the other part implemented as part of Visio. The following sections describe the low-level design for all the blocks mentioned in Section V.B.

Figure V.4: M2Code Low Level Design
V.D.1 UserInterface

As described in Section IV.B.1, Visio is a stand-alone diagramming tool, that supports its programmatic control via COM interfaces. Figure V.4 shows the UserInterface consisting of two blocks. One part of Visio and the other part of M2Code. In order to provide user input for generating state machines, we used Visual Basic for Applications (VBA), which is a rapid application development tool. Using VBA, we designed different forms for displaying all the components and MSCs in the current project so that the user can select the components and the corresponding MSCs that need to be converted to state machines. Additionally, as described in Section IV.B.1, Visio provides a very powerful graphical canvas interface and allows pre-configured widgets called stencils to be placed on this canvas and connect them.

Although, we don’t have any visibility to the exact implementation of Visio, we can describe it in the form of a MVC design pattern. The different diagrams on Visio, can be thought of as the View, the internal representation of the diagrams (the list of the contained shapes, their coordinates, their connectivity etc.) can be thought of as the Model and the entity that takes the user input and manages the tool as the Controller. As the user is entering this information in Visio, it responds appropriately by updating its views. In addition to this, our tool M2Code would also like to interpret these to a level of detail higher than just a set of shapes and interconnections but rather using the semantics and notation of Basic and High-Level MSCs as described in Section II.C. This also allows us to impose constraints and perform validations on the MSCs as they are being drawn by the user.

In order to build the internal model within our tool as the diagrams are being drawn on Visio, we need to implement a UserInterface block within our tool also. This is represented by the second block labeled UserInterface(M2Code) in Figure V.4. The design within this module also follows the MVC design pattern. Instead of the Controller responding to user input directly from the user,
it responds to event notifications received from Visio via the COM event notification framework. When these events are received, this Controller builds up the internal representation to reflect the diagrams within Visio but only pertaining to the information needed by M2Code. Once all the MSCs have been input by the user and the command for generating state machines is invoked, the Controller and Model play the additional role of the ModelConverter, wherein the conversion algorithms are applied to Model and the resulting state machines are sent to the StateMachineMinimizer for minimization and to the LayoutGenerator for calculating layout information for the generate diagrams. This layout information is used by the View within M2Code to display these diagrams on Visio.

Figure V.5: M2Code Visio Interface Diagram

Figure V.5 shows the class diagram for the UserInterface related abstractions within M2Code. The class CVisionModel, a singleton instance, represents the
current model under design and contains references to the current Visio document that is being modified via the IVDocument interface pointer. CVisioModel also contains a collection of CVisioDiagram instances, which encapsulate each of the diagrams (MSCs, state machines and SSD) via the IVPage interface pointer and a collection of CVisioComponent instances, each of which represent a component in the system under design. The class CVisioDiagram is a generalization for all the different types of diagrams that can be drawn by our tool and contains a collection of CVisioShape instances each representing the different shapes that can be drawn in that particular diagram via the IVShape interface pointer. The class CVisioShape is a generalization for all the different shapes that can be drawn by our tool and contains a collection of shapes that it is connected to. Since a component can appear in more than one diagram in the form of an axis, CVisioComponent contains reference to one or more CVisioShapeAxis instances.

![Sequence diagram](image)

**Figure V.6: M2Code UserInterface Add/Remove Diagrams**

The sequence diagram in Figure V.6 describes the scenarios of adding and removing diagrams. Once a new diagram is created, the user can add/remove
shapes and connect them via the different symbols relevant to the type of diagram being edited. This scenario is described in Figure V.7.

If this is a saved document, iterate through all the shapes on the page and call `addShape()` on each of them.

User adds a new shape from the stencil.

User deletes a shape from the diagram.

User adds a connector between shapes in the diagram.

User deletes connections between shapes in the diagram.

Figure V.7: M2Code User Interface Add/Remove/Connect Shapes

V.D.2 ModelConverter

In order to represent MSCs, we designed a class hierarchy to mirror all the elements of Basic and High Level MSCs as described Section II.C.1. A small subset of this is shown in Figure V.8. The base class for all the MSC related
abstractions is InteractionElement. From this class, we derive two types classes - Atom and Compound. Atom’s are used to represent basic elements that cannot be decomposed further, such as Ref, StateMarker and Message. Compound’s, on the other hand can contain one or more Compartment’s, each of which can contain one or more InteractionElements. For example, a Par object (representing a PAR operator) can contain two or more compartments, each of which can contain one or more messages, state markers or references to other BMSCs or HMSCs. We utilized the composite design pattern for these types of interaction elements.

![Diagram of M2Code MSC Domain Model](image)

**Figure V.8: M2Code MSC Domain Model**

The MSC domain model is constructed by M2Code as the user is inputting the diagrams on Visio. Once all the MSCs have been input, they can be converted into state machines by selecting the components to be converted from the UserInterface and invoking the convert operation as described in the sequence diagram shown in Figure V.9. The conversion process starts at each CVisioComponent, which iterates through each of its instances in the diagrams that need to
be converted. The actual algorithm for the conversion process is described in detail for each type of operator in Chapter VII.

V.D.3 StateMachineMinimizer

Once the ModelConverter generates the component level state machine, it needs to invoke the StateMachineMinimizer to remove all the $\epsilon$ transitions and minimize the generated state machines. This module takes component level state machines, generated by the ModelConverter as input and produces an output that represents the deterministic finite automaton for that component. Rather than design and implement this entire module, we decided to utilize the finite automata library FA as mentioned in Section IV.B.4. However, since our tool is implemented in c++, we needed to implement an interface layer between the ModelConverter and this library. To solve this problem we wrote a lightweight wrapper M2CodeToFA in Java that performs the translations between these modules. The output from the ModelConverter was converted into text files and provided as in-
put to this wrapper and after the minimization process, this wrapper would store
the result into another text file. The input file consisted of a newline terminated
list of state transitions of the form \([s0, \text{!m}, s1]\), where \(s0\) is the initial state, \(m\)
the message causing the transition and \(s1\) the final state after the transition. The
format of output text file from this wrapper was the same as the format needed by
the next module - LayoutGenerator. This format is described in detail in IV.B.5.
We packaged this wrapper into a jar file (automata.jar) and provided it a manifest
file so that it could be invoked from command line as follows:

```
java automata.jar -i initial -f final -s filename -o filename -a -m
```

The -i option was used to add an initial state, -f for final state, -s for
the file name of the state machine to be minimized, -o for the file name to store
the result, -a to enable arrowheads during layout, -m to generate output file after
minimization. The -i, -f and -s options could be repeated any number of times
if need be. The -a option was added so that the arrowhead could be optionally
turned on or off, allowing some flexibility of usage based on the requirements of
the next tool in the framework. The -m option was added for debugging purposes
so that the automaton before and after minimization could be viewed.

V.D.4 LayoutGenerator

After the ModelConverter invokes the StateMachineMinimizer, it needs
to display the state machine as a diagram on Visio. Visio provides some basic
layout functionality, which did not serve all our needs. Hence, as mentioned in
Section IV.B.5 we decided to use DOT as a stand alone executable and invoked
it from M2Code. The output from the StateMachineMinimizer is directly fed as
input to DOT and we instruct it to calculate layout information in text format,
so that we can process the layout information from our tool. In order to parse
this output file from DOT a string tokenizer class was implemented. This c++
class, M2Document parses a text file into lines and words. The class hierarchy
for this tokenizer is as shown in Figure V.10. This tokenizer is invoked and by CVisioDiagramSC (for displaying state machines) and CVisioDiagramSSD (for displaying SSDs). Once each of these classes process the tokenized file, they call the DrawOval and DrawSpline interfaces on the IVPage Visio interface to present the diagram to the user via Visio. The sequence diagram in Figure V.9 shows the flow of control to display a generated SC on Visio after minimizing it.

![Diagram](image)

Figure V.10: Text file tokenizer

V.D.5 ModelExporter

After the MSCs are converted to state machines, the user can run tools to simulate the generated model in order to verify if the generated model exhibits the desired behavior. If it does, then the generated state machines can be converted into executable code. In order to accomplish this, we decided to utilize other tools for simulation and code generation. For simulation, we decided to use AutoFocus [1] and RTCGen [7] for code generation. M2Code interfaces to these tools by exporting the generated model into a format that can be imported by them. Further processing is carried out on those tools. In order to export the model, M2Code first builds up a domain model for the generated state machines and then parses through it to generate the necessary files. The details of the two tools, their inputs and the domain model are described in detail in Chapter VI.
Chapter VI

Interface to other Tools

VI.A Introduction

In Chapter I, we described our proposed solution as a tool that could generate implementation details from interaction specification. Additionally, we also described that in order to add extra features to the tool, it would be desirable to interface it with other tools that provide value added behavior, such as simulation and code generation using the generated models. In Chapter III we described import and export requirements from such a tool. In order to meet the requirements III.D2 and III.D3, we need to be able to export the models from M2Code to AutoFocus and RTCGen. This chapter describes these two tools and how we interface M2Code to both of them.

VI.B AutoFocus

AutoFocus is a tool developed by the Department of Informatics, Technische Universität München [1]. It is a Java based graphical modeling tool that is used for the specification and analysis of distributed systems. AutoFocus provides a set of graphical editors for drawing System Structure Diagrams (SSDs) and State Transition Diagrams (STDs) and a rich set of simulation tools called Extended Event Traces (EETs) that can be invoked once the model is checked for
consistencies. SSDs, as described in Section I.B.1, are high level diagrams that describe the different components in the system and the channels of communication between them. STDs, on the other hand as described in Section I.B.1 are behavioral diagrams that describe the various states of the system and the transitions between them. These diagrams describe the system to a fine grain of detail and hence are very close to the actual implementation of the system. For large and complex systems, due to feature interactions and asynchronous message exchange, these diagrams can get fairly complicated due to state space explosion.

![Diagram](image1)

Figure VI.1: AutoFocus project creation and DTD definition

In order to understand the tool better, we will walk you through the usage of the tool to model an example from the automotive domain consisting of two components KF (Key Fob) and CONTROL (Controller). In this example, these two components exchange messages over channels KC and CK and exchange messages `lck`, `unlk` and `ok`. KF initiates the message transaction by sending either a `lck` or an `unlk` message to CONTROL. In response to this message, CONTROL
sends an *ok* message. In order to use AutoFocus for modeling this system, the user would first create a new project on AutoFocus as shown in Figure VI.1a. Creation of the project causes four sections to be automatically created under the project - SSD, STD, EET and DTD.

The next step consists of defining the data types for this model by editing the Data Type Definition (DTD). In this case, we define a type called MSGS and use the same type for all three messages - *lck*, *unlk* and *ok* as shown in Figure VI.1b. Next, we need to define the two components in the system KF and CONTROL and their channels of communication. This is accomplished by drawing the SSD as shown in Figure VI.2a.

Figure VI.2: AutoFocus Usage Demo - SSD for the system and STD for KF

Next, we need to describe the automaton for both these components by editing their STDs. For KF, the automaton consists of three states - IDLE, LOCKING and UNLOCKING. It is initially in the IDLE state. It moves to the LOCKING state by sending the message *lck* on channel KC. Upon receipt of the *ok* message on channel CK, it returns back to the IDLE state. When it sends the *unlk* message on channel KC, it moves to the UNLOCKING state. Finally, when
it receives the *ok* message on channel CK, it returns back to the IDLE state. This is shown in Figure VI.2b. Similarly, we define an automaton for CONTROL as shown in Figure VI.3a. After drawing all these diagrams, the user can invoke the tool to verify the consistency of the model and simulate it. For this example, the execution trace generated is shown in Figure VI.3b.

![Diagram](image)

(a) (b)

Figure VI.3: AutoFocus Usage Demo - STD for CONTROL and EET for the system

In the above example, since there are only two components and only a few transitions, it is fairly straightforward for the designer to think of the different states and the transitions and draw them on the editor. However, as described in Section I.D, we propose that it is easier for users to describe the requirements of the system in the form of scenarios that should and should not occur in the system and have the tool carry out the process of converting this into state machines. Once these state machines are generated, it would definitely be beneficial to the user if the execution traces of the generated system could be simulated. Since AutoFocus has a powerful simulation tool, we realized that rather than design one
from scratch, it would be better if we could import M2Code generated models into AutoFocus and simulate these models.

In order to accomplish this, a couple of input formats of AutoFocus were explored. The version of AutoFocus that we used did not utilize an XML DOM parser and hence depended on the exact position of the various XML tags in the input model file. This made it very hard as determining the exact positions including indentation of all the fields was not a trivial solution. It supported another format called QML, which was an internal format that it used to load/store models. This format is very hierarchical and describes the various components, the channels of communication, the different states and the transitions between these states. Additionally, the actual position of these elements had to be provided as these would be used to draw the various elements in the graphical editor of AutoFocus. In order to easily export the M2Code generated models, we designed a domain model, which is discussed in more detail in Section VI.D. AutoFocus can also export models (that are drawn using its editor or imported from another tool) in XML format. This format, however, consists of only the model elements, i.e. components, channels, ports etc and does not contain any graphical information such as the layout information of each of these model elements.

VI.C RTCGen

The RT Code Generator (RTCGen) [7] [6] is designed and implemented at the S3E Lab at UCSD. It is designed to take component level automata as an XML file and convert it into executable code for the RT CORBA middleware. The RTCGen tool was being worked on prior to the initial M2Code design and hence, the tool used the XML output generated by AutoFocus as the input file. Once M2Code was implemented, the integration efforts between M2Code and RTCGen were initiated and instead of rewriting the XML input format for RTCGen, it was decided to use the same format as the one produced by AutoFocus. This
commonality made it easier to come up with a domain model for representing the minimized automata generated by M2Code. Section VI.D describes this domain model in detail. The XML file that is exported by M2Code resembles the QML file that is exported for AutoFocus except that it does not contain information regarding the graphical layout of the various components. This is due to the fact that, unlike AutoFocus, RTCGen does not have a graphical editor or a graphical display that can visually show the model that is being converted into code. However, it does need the information regarding the components, ports, channels of communication, states and their transitions. Hence, the export format is slightly different but the core information that they carry is the same.

VI.D State Machine Domain Model

![Diagram of State Machine Domain Model - Project and Components](image)

Figure VI.4: State Machine Domain Model - Project and Components

The motivation behind creating a domain model for the state machine was that we needed to represent the generated minimized automata in a form that would allow easy traversal of the various abstractions and a form that would easily map to the input file formats of both AutoFocus and RTCGen. The following class diagrams show the various classes, their hierarchy and their relationships. The key
abstractions are described below.

VI.D.1 Models

In order to better understand the role of each of the classes shown in Figures VI.4 and VI.5, we use the SSDs and STDs that were described in the example in Section VI.B. Using the domain model described in this section, the SSD in Figure VI.2a can be described as follows. The Repository would contain an instance of this Project - “AutoFocusUsageDemo”. This project would contain one overall Component - Main to represent the entire system. This component would contain references to the DTD being used for all the elements contained in this component. It would have references to two Channels - KC and CK to represent the channels of communication between the subcomponents - Control and KF. We will now describe the relationships of the classes that are related to the Component class as shown in Figure VI.5 with respect to component KF. This component would have two ports - KC0 a source port for channel KC and CK1 a destination port for channel CK. Figure VI.6a shows these relationships.

The component instance for KF would also contain a reference to its Automaton. This automaton, would contain a reference to the a State - “Protocol-KF” which, is a super state and contains all the sub-states in that automaton. This state “Protocol-KF” consists of references to TransitionSegments, each instance of which, represents one of the transitions in the STD for that component. In the STD for the KF component described in Figure VI.2b, there are four TransitionSegments - lck, ok, unlk and ok. Note that there are two transitions with the message ok both originating from the IDLE state but one terminating in the LOCKING state and the other in the UNLOCKING state. Each of these transition segments are associated with two InterfacePoints - one on each end of the transition. These transitions are also associated with either Input or Output ports, thereby describing how messages being sent from components via channels enter the component from the associated port and cause a transition in the state of that
Figure VI.5: State Machine Domain Model - Components and Automaton
component to occur. InterfacePoints connect to TransitionSegments on one end (as described above) and to State objects on the other end. Hence, every State object also has InterfacePoints for every transition that occurs into or out from itself. Figure VI.6b shows these relationships.

VI.D.2 Views

Graphical editors need to keep track of the physical position (in terms of coordinates) of all the visual elements in the diagram. This position information is not only needed when the model is being edited but also when it is saved to a file so that the diagram can be recreated to its original state when it is loaded back into the editor again at a later point in time. All of the elements that need to be displayed on the graphical editor - Components, Channels, States, TransitionSegments and InterfacePoints need to be associated with one or more Views as shown in Figure VI.7. Since the actual type of information that is needed for each graphical element is different, the View class serves as the base class for all the different types of views as shown in Figure VI.8. These class diagrams describe the View class, its derived classes and the relationships between the other abstractions.
Figure VI.7: State Machine Domain Model - View base class

Figure VI.8: State Machine Domain Model - View derived classes
Chapter VII

Algorithms

In Chapter II, we described the semantics of Basic and High Level MSCs and provided an intuition of how MSCs can be converted into state machines. In Chapter V, we described the architecture of the tool and how the various components of the tool gather the necessary information from the Basic and High Level MSCs and invoke the conversion algorithms. This chapter explains the implementation of the algorithms described in [23] that are used by the our tool to convert MSCs into state machines. This chapter attempts to explain the approach for implementing the algorithms described in the thesis mentioned above by using examples followed by a set of steps that need to be carried out for the conversion process.

VII.A Conversion of a Basic MSC to a State Machine

Figure VII.1 a shows a Basic MSC consisting of two components A and B. It shows a set of interactions consisting of message exchange between the two components along with state markers showing states that the user expects to see the component in. For example, after component A receives message $m1$ from component B, it sends message $m2$ to component and moves to state $s4$. The algorithm to convert component A into a state machine is described below. The conversion algorithm for component B is identical to that of A and hence not
described in detail.

Figure VII.1: Basic MSC and converted State Machine for Components A and B

1. Create an empty state machine scA to represent the converted state machine for component A.

2. Add temporary states around all the messages sent and received by component A. For example, add states \(_s0_\) and \(_s1_\) around message \(m0\); states \(_s2_\) and \(_s3_\) around \(m1\) and so on.

3. Start from the first MSC element present on the axis timeline of the component and convert as follows. For every user-defined state marker, create an \(\epsilon\) transition between the state marker and the following temporary state
marker. For every message, create a state transition by using the temporary 
state marker before the message as the initial state and the temporary state 
marker after the message as the final state. For example, state marker s0 gets 
converted as a state transition [s0, ε, s0] and message m0 gets converted 
into a state transition as [s0, !m0, s1].

4. Add all the state transitions generated above into scA.

5. Pass the resulting state machine scA through a minimizer and the result 
represents the converted state machine for component A as shown in Figure 
VII.1b. Repeat the same set of steps for component B and the generated 
state machine for component B will be as shown in Figure VII.1c.

VII.B ALT

The MSC element ALT is used to represent alternative flow of control. 
Figure VII.2a shows example usage of an ALT operator in a Basic MSC. In this 
Basic MSC, after execution of message m0, and depending on the result of the 
expression being evaluated, either the messages labeled m2 and m3 would be 
exchanged between Component A and B or the messages labeled m4 and m5 
would be exchanged between Component A and B. The message m1 would be 
exchanged irrespective of the result of expression evaluation. The algorithm for 
the conversion of this ALT operator into a state machine is as explained below. 
The details of converting the messages themselves is explained in Section VII.A. 
Also, the description covers conversion of only Component A as the conversion of 
Component B is very much similar to the steps described for Component A.

1. Create three empty state machines scAlt1, scAlt2 to represent the converted 
state machine for each of the alternatives and scAlt to represent the state 
machine for the entire Basic MSC.

2. Add temporary states s0, s1 and s2 sur-
Figure VII.2: ALT operator in Basic MSC and converted state machine

rounding the first condition of the ALT element and \( s_3 \) and \( s_4 \) surrounding the second condition of the ALT element and \( s_5 \) just after the ALT element.

3. Assign \( s_0 \) as the initial state of scAlt1 and scAlt2 so that any elements before the ALT element would cause a transition to \( s_0 \).

4. Assign \( s_5 \) as the final state of scAlt1 and scAlt2 so that after either of the two alternatives, there would first be a transition to \( s_5 \), after which there could be transitions to any elements following the ALT operator.

5. Convert the messages within the first ALT condition (exchange of messages \( m_2 \) and \( m_3 \)) as per the steps mentioned in Section VII.A. Let the resulting transitions be T1. Now, add \( \epsilon \) transitions from \( s_0 \) to \( s_I \), \( s_I \) to T1, T1
to \( s_2 \) and \( s_2 \) to \( s_5 \). Add T1 and all the \( \epsilon \) transitions to scAlt1. Now, the state machine scAlt1 represents the state machine for the messages in the first compartment of the ALT.

6. Similarly, convert the messages within the second ALT condition (exchange of messages \( m_4 \) and \( m_5 \)) as per the steps mentioned in Section VII.A. Let the resulting transitions be T2. Now, add \( \epsilon \) transitions from \( s_0 \) to \( s_3 \), \( s_3 \) to T2, T2 to \( s_4 \) and \( s_4 \) to \( s_5 \). Add T2 and all the \( \epsilon \) transitions to scAlt2. Now, scAlt2 represents the state machine for the messages in the second compartment of the ALT.

7. In order to obtain the state machine for both the alternatives combined, assign all the initial states of scAlt1 and scAlt2 as the initial states of scAlt and all the final states of scAlt1 and scAlt2 as the final states of scAlt. Add all the transitions within scAlt1 and scAlt2 into scAlt.

8. Convert \( m_0 \) as per the steps mentioned in Section VII.A. Let the resulting transitions be T3. Now, add \( \epsilon \) transitions from T3 to \( s_0 \). Add T3 and the epsilon transitions to scAlt.

9. Convert \( m_1 \) as per the steps mentioned in Section VII.A. Let the resulting transitions be T4. Now, add \( \epsilon \) transitions from \( s_5 \) to T4. Add T4 and the \( \epsilon \) transitions to scAlt.

10. Pass the resulting state machine through a minimizer and the result represents the converted MSC as shown in Figure VII.2b.

VII.C PAR

The MSC operator PAR is used to express parallelism in the sequence of execution of messages. In the MSC shown in Figure VII.3a, messages \( m_2 \) and \( m_3 \) need to occur in the order specified, i.e., \( m_2 \) must always occur before \( m_3 \). Similarly, \( m_4 \) must always occur before \( m_5 \). However, the relative ordering of these
groups of messages can be different, leading to a wide variety of combinations. For example, the sequence of messages could be \(!m2, ?m3, ?m4, !m5\) or \(!m2, !m4, ?m3, ?m5\) or \(!m2, !m4, ?m5, ?m3\) and so on. The algorithm for conversion of component A with the PAR operator is described below.

Figure VII.3: PAR operator in Basic MSC and converted State Machine

1. Add temporary states \(_s0_\) and \(_s1_\) surrounding the PAR operator.

2. Convert the messages within each compartment into state machines using the steps mentioned in Section VII.A. Let the resulting state machines for the first compartment be \(sc\text{Par1}\) and for the second compartment be \(sc\text{Par2}\).
Create an empty state machine scPar to hold the resulting state machine for the entire PAR operator.

3. Iterate through each of the states in scPar1. For each of these states $s_1$, iterate through each of the states $s_2i$ in scPar2 and find each transition $t$ that originates from $s_2i$ and has a final state of $s_2f$ of the form $[s2i, m2, s2f]$. Create new transition in scPar which originates from a new state $s_1s_2i$ but ends in $s_2f$ of the form $[s1s2i, m2, s2f]$. Add all these states and transitions to scPar.

4. Perform the same step as above but this time reverse the order of iteration by first iterating over all the states in scPar2, followed by iterating over all the states in scPar1. This step will generate transitions of the form $[s2s1i, m1, s1f]$. Add all these states and transitions to scPar.

5. Add an $\epsilon$ transition between $s_0$ and the new state that is created by combining the initial states of scPar1 and scPar2. This transition will connect the state machine for the PAR operator with the generated state machine for any BMSC operators that occur prior to it in the time line of the component.

6. Add an $\epsilon$ transition between the new state that is created by combining the final states of scPar1 and scPar2 and $s_L$. This transition will connect the state machine for the PAR operator with the generated state machine for any BMSC operators that occur after it in the time line of the component.

7. Pass the resulting state machine through a minimizer and the result represents the converted MSC as shown in Figure VII.3b.

**VII.D LOOP**

The Basic MSC operator LOOP is used to describe interactions that have a set of messages repeatedly exchanged and the number of iterations that occur on the loop is controlled by an invariant. Figure VII.4 shows a Basic MSC consisting
of two components A and B with a LOOP operator. In this example, messages \( m0 \) and \( m1 \) are repeatedly sent from A to B until the loop exits. The algorithm for conversion of component A with the LOOP operator is explained below.

1. Add temporary states \(_s0_\), \(_s1_\), and \(_s2_\) just before the LOOP operator, \(_s3_\) and \(_s4_\) surrounding the compartment of the LOOP operator and \(_s5_\) just after the LOOP operator.

2. Convert all the messages in the compartment of the LOOP element as per the algorithm mentioned in Section VII.A into state machine scLoop.

3. Assign \(_s1_\) as the initial state and \(_s2_\) as the final state of scLoop.

4. Add an \( \epsilon \) transition between \(_s0_\) and \(_s1_\). This transition will serve as the entry criteria for the loop.

5. Add an \( \epsilon \) transition between \(_s2_\) and \(_s3_\). This transition will serve as the exit criteria for the loop.

6. Pass the resulting state machine through a minimizer and the result represents the converted MSC as shown in Figure VII.4b.

Figure VII.4: LOOP operator in Basic MSC and converted State Machine

(a) 

(b)
VII.E JOIN

The BMSC operator JOIN is used to combine BMSCs in such a way that they synchronize over common messages and interleave over the remaining. Figure VII.5a shows a BMSC consisting of two components A and B with a JOIN operator. Here, the JOIN operator is used to describe the combination of the messages in the upper compartment with that in the lower compartment. In this example, these compartments share a common message \( m0 \). These two compartments need to be joined keeping this common message in mind. The exchange of the other messages can be interleaved as long as their respective ordering is maintained. For example, both messages \( m3 \) and \( m2 \) must occur before message \( m0 \) but they could occur in any order - message \( m3 \) before message \( m2 \) or vice versa. The algorithm for conversion of component A with the JOIN operator is described below.

1. Using the algorithm described in Section VII.A, convert the messages in each of the compartments into state machines - scJoin1 and scJoin2 respectively. Create an empty state machine scJoin to represent the resulting state machine for the all the messages in the JOIN operator.

2. Check if the two state machines share any common messages. If not, use the algorithm for the PAR operator to combine these two state machines and add all the states and transitions to scJoin. If they do share common messages, proceed with the following steps.

3. For the state machine scJoin1, build sub-state machines by enumerating all the paths originating from its initial state and either terminating at its final state or a state from which there are no outgoing states. After this step, say there are \( p \) such sub-state machines.

4. Repeat the above step on state machine scJoin2. After this step, say there are \( q \) such sub-state machines.

5. Now, iterate through each of the \( p \) sub-state machines and for each of these
Figure VII.5: JOIN operator in Basic MSC and converted State Machine

sub-state machines, iterate through the q sub-state machines looking for common messages. If neither of these sub-state machines share a common message, perform the PAR operation on them. If they do share common messages, follow the steps below.

6. Let the i-th iteration of the sub-state machines of scJoin1 be $p_i$ and the j-th iteration of the sub-state machines of scJoin2 be $q_j$. Start from the initial state on both $p_i$ and $q_j$ and keep walking through the state transitions until a common message is reached on both of them.

7. The common message could be the same message or different messages. For example, the transition could be $[s1, m_x, s2]$ on $p_i$ and $[s3, m_x, s4]$ on $q_j$. 
or \([s_1, m_x, s_2]\) on \(p_i\) and \([s_3, m_y, s_4]\) on \(q_j\) where \(m_x\) and \(m_y\) are common messages between \(p_i\) and \(q_j\).

8. If the common message is the same, then perform the PAR operation on all the transitions prior to the common one and add the resulting interleaving to scJoin. Then add one transition with the common message of the form \([s_1s_3, m_x, s_2s_4]\) to scJoin. After this, proceed with walking through the transitions as described above, looking for common messages and apply the same set of steps when they are found.

9. If the common message is not the same, then the two state machines cannot be combined as neither of the transitions in the two state machines can proceed to the next state as each of them is waiting for the other to reach the same common message. In this case, return an empty state machine.

10. Pass the resulting state machine through a minimizer and the output represents the converted MSC as shown in Figure VII.5b.

VII.F REF

The Basic MSC operator REF is used to reference a Basic MSC from within another Basic MSC. This allows the user to compose Basic MSCs sequentially. References are analogous to function calls in the programming paradigm, wherein the normal flow of messages in the interaction specification are executed until the point of the REF operator on the timeline. When a REF operator is encountered, the interaction flow now switches to the set of interactions described in the referenced Basic MSC. Once all the interactions from that MSC are executed, control is transferred back to the original MSC and the execution continues with the interaction following the REF operator.

Figure VII.6a describes a Basic MSC MSC-X consisting of two components A and B with a REF operator that references Basic MSC MSC-Y, shown in
Figure VII.6b. In this example, the interactions start from MSC-X, where component A sends message $m_0$ to B, then control is transferred to MSC-Y. After completion of interactions in MSC-Y, control is returned to MSC-X and it proceeds by component A awaiting receipt of message $m_1$ from component B. The algorithm for conversion of component A with the REF operator is explained below.

1. Convert the message $m_0$ into a state machine by adding temporary states $s_0$ and $s_1$ around it.

2. Add temporary states $s_2$ and $s_3$ around the REF operator.

3. Convert the messages in MSC-Y into a state machine using the steps mentioned in Section VII.A. Let the resulting state machine be $sc_{\text{Ref}}$.

4. Add $\epsilon$ transitions from $s_2$ to the initial state of $sc_{\text{Ref}}$ and from the final
state of scRef to $s_3$.

5. Convert the message $m1$ into a state machine by adding temporary states $s_4$ and $s_5$ around it.

6. Finally, add $\epsilon$ transitions from $s_1$ to $s_2$ and $s_3$ to $s_4$. This step connects the generated state machine for MSC-Y to that of MSC-X.

7. Pass the resulting state machine through a minimizer and the output represents the converted MSC as shown in Figure VII.6c.

**VII.G PREEMPT**

The Basic MSC operator PREEMPT is used to describe interactions that can cause exceptions to occur, i.e. the occurrence of one or more messages causing the current execution to terminate and the transitions in the exception handler to get invoked. Figure VII.7 shows a Basic MSC consisting of two components A and B with a PREEMPT operator. In this example, the interactions in the first compartment, i.e. exchange of messages $m_0$, $m_1$ and $m_2$ represent the normal behavior. However, if message $m$ occurs at any point during the exchange of these messages (on a predefined preempt channel), the current transition needs to complete, but rather than proceeding with the next transition in the normal behavior, the interaction needs to terminate after the exchange of message $m_3$. The algorithm for conversion of component A with the PREEMPT operator is explained below.

1. Add temporary states $s_0$ and $s_1$ surrounding the first compartment of the PREEMPT element and $s_2$ and $s_3$ surrounding the second compartment of the PREEMPT operator.

2. Convert all the messages in the first compartment of the PREEMPT operator as per the algorithm mentioned in Section VII.A into state machine scPreempt1.
3. Assign \( s_0 \) as the initial state and \( s_1 \) as the final state of \( \text{scPreempt1} \).

4. Convert all the messages in the second compartment of the PREEMPT operator as per the algorithm mentioned in Section VII.A into state machine \( \text{scPreempt2} \).

5. Assign \( s_2 \) as the initial state and \( s_3 \) as the final state of \( \text{scPreempt2} \).

6. Create an empty state machine \( \text{scPreempt} \) and add all the transitions in \( \text{scPreempt1} \) and \( \text{scPreempt2} \) to \( \text{scPreempt} \).

7. Create transitions with all the preemptive messages (\( m \) in this example) on the predefined preempt channel \( \text{PA} \) between all the states of \( \text{scPreempt1} \) to all the initial states of \( \text{scPreempt2} \) (\( s_2 \)).
8. Finally, set the initial states of scPreempt to the initial states of scPreempt1 ($s0_-$) and the final states of the scPreempt to the final states of scPreempt1 ($s1_-$) and scPreempt2 ($s3_-$).

9. Pass the resulting state machine scPreempt through a minimizer and the output represents the converted MSC as shown in Figure VII.7b.

**VII.H TRIGGER**

The Basic MSC element TRIGGER is used to describe interactions where occurrence of one or more messages causes a trigger to occur, leading to the eventual execution of one or more messages. Figure VII.8a shows a Basic MSC consisting of two components A and B with a TRIGGER operator. In this example, the exchange of messages $m0$, $m1$ and $m2$ between A and B is expected to trigger the exchange of messages $m3$ and $m4$. The algorithm for conversion of this MSC operator for component A is explained below.

1. Add temporary states $s0_-$ and $s1_-$ surrounding the first compartment of the TRIGGER operator and $s2_-$ and $s3_-$ surrounding the second compartment of the TRIGGER operator.

2. Convert all the messages in the first compartment of the TRIGGER element as per the algorithm mentioned in Section VII.A into state machine scTrigger1.

3. Assign $s0_-$ as the initial state and $s1_-$ as the final state of scTrigger1.

4. Convert all the messages in the second compartment of the TRIGGER operator as per the algorithm mentioned in Section VII.A into state machine scTrigger2.

5. Assign $s2_-$ as the initial state and $s3_-$ as the final state of scTrigger2.
6. Create an empty state machine $sc_{Trigger}$ and add all the transitions in $sc_{Trigger1}$ and $sc_{Trigger2}$ to $sc_{Trigger}$.

7. Create $\epsilon$ transitions between all the final states of $sc_{Trigger1}$ ($s1$) to all the initial states of $sc_{Trigger2}$ ($s2$) and add these transitions into $sc_{Trigger}$.

8. Create $\epsilon$ transitions between all the final states of $sc_{Trigger2}$ ($s3$) to all the initial states of $sc_{Trigger1}$ ($s0$) and add these transitions into $sc_{Trigger}$.

9. Set the initial states of $sc_{Trigger}$ to the initial states of $sc_{Trigger1}$ and the final states of $sc_{Trigger}$ to the final states of $sc_{Trigger2}$.

10. Find all the states that have transitions to the final states of $sc_{Trigger1}$ and set those states also as final states in $sc_{Trigger}$. This ensures that the final state could be reached prior to the completion of the last message in the first compartment of the TRIGGER (message $m1$ in this example). However, if
the transition $m2$ occurs, then all the transitions of the second compartment must eventually be executed.

11. Pass the resulting state machine scTrigger through a minimizer and the output represents the converted MSC as shown in Figure VII.8b

VII.1 Conversion of an HMSC into BMSC terms

HMSCs, as described in Section II.C.1, are used to compose other HMSCs or BMSCs into one interaction specification. The MSCs to be composed are described using the REF operator and the composition itself is described using different types of connectors as described in Section II.C.1. The HMSC shown in Figure VII.9a, describes the composition of four MSCs - A, B, C and D. The composed interactions start with those described in A and end with those described in D. After completion of execution of interactions in A, they proceed with those described in B after which, there is a choice to either proceed with C or D. If C is chosen, then after completion of C, the interactions described in A repeat again. If D is chosen, then after completion of interactions in D, there is again a choice to either end the interaction specification or go back to B.

Since HMSCs are used to compose other MSCs together, they can be converted to state machines by converting the entire HMSC into a set of operators that can be applied over the MSCs that are being composed. For example, the choice from MSC B to either MSC C or MSC D in Figure VII.9a, can be interpreted as a BMSC starting from a REF to MSC B followed by the ALT operator consisting of REF to MSC C as the first compartment and REF to MSC D as the second. So, conversion of HMSCs to state machines first involves conversion into a set of BMSC terms, followed by conversion of the referenced MSCs into state machines and finally application of the BMSC terms to the referenced state machines. The algorithm for converting an HMSC into a BMSC term is described below.

1. Transform the HMSC into an intermediate automaton form for further pro-
cessing by replacing each START operator by a state and mark it as the initial state. Replace each REF operator with a state and create a transition between each state corresponding to the connection between the REF operators. Label these transitions using the name of the MSC that is the destination of the connection. For the example shown in Figure VII.9a, the intermediate automaton is shown in Figure VII.9b.

2. In the intermediate automaton, mark all the states that lead to the END operator as the final state of the automaton.

3. Add a new state $s_t$, called the trap state, to the intermediate automaton. We will describe the use of this state in the steps that follow.

4. Iterate through all the states (except for the initial state and the trap state) and eliminate them one by one, using the rules described below. Categorize the transitions that are related to the state being eliminated as either incoming or outgoing or looping based on where the transition begins and ends.
Transitions that end in a state are incoming, transitions that begin from a state are outgoing and transitions that have the state as both begin and end are looping.

- If there are more than one looping transitions, combine them into one LOOP term consisting of ALT terms for each of the looping transitions. Figure VII.10a shows a sample state $s_n$, that needs to be eliminated, consisting of two loops W and X. These two loops can be combined into one loop as shown in Figure VII.10b.

- If there are no outgoing transitions, iterate through each of the incoming transitions and create a new transition as an SEQ term consisting of the label on the incoming transition and the loop transition (if present). The source for this transition is the source to the incoming transition and the destination is the trap state $s_t$. In the sample shown in Figure VII.10b, if the outgoing transitions Y and Z were absent, and if state $s_n$ were to be eliminated, then there would be one new transition to the trap state $s_t$ with the term SEQ(V, LOOP(ALT(W, X))).

- If outgoing transitions exist, then iterate through each of the incoming transitions and create a new transition as a SEQ term consisting of the label on the incoming transition and the loop transition (if present) and each of the outgoing transitions individually. The source for each
of these transitions is the source to the incoming transition and the
destination is the destination of the outgoing transition that it is be-
ing combined to. For the sample shown in Figure VII.10b, if state
$s_n$ were to be eliminated, then the new transitions would be $\text{SEQ}(V, \text{LOOP}(\text{ALT}(W, X)), Y)$ and $\text{SEQ}(V, \text{LOOP}(\text{ALT}(W, X)), Z)$.

5. Once all the states (except the initial state and the trap state) are eliminated, there will be one or more transitions between these two states, added as a consequence of the elimination process. Combine all these transitions into one using the ALT term. The final result represents the HMSC in the form of BMSC terms and hence can be converted to an SC by plugging in the SC for each of the MSCs and applying the generated BMSC operators.

If we apply the above set of steps to the example shown in Figure VII.9a, the resulting output looks like:

```
[ Start10,
  \text{SEQ}((\text{ALT}((\text{SEQ}((\text{SEQ}((?\text{Ref}0)(?\text{Ref}1)))(?\text{Ref}3)))(?\text{Ref}3)))
  (\text{LOOP}<\text{inf}>:((\text{ALT}((\text{SEQ}((?\text{Ref}1)(?\text{Ref}3)))))(?\text{Ref}3))
  (\text{SEQ}((\text{SEQ}((?\text{Ref}1)(?\text{Ref}2)))(\text{LOOP}<0,\text{inf}>:((\text{SEQ}((?\text{Ref}0)
  (?\text{Ref}1)))(?\text{Ref}3)))))(?\text{Ref}3)))))
  (\text{SEQ}((\text{SEQ}((?\text{Ref}0)(?\text{Ref}1)))(?\text{Ref}3))))
  (\text{SEQ}((?\text{Ref}0)(?\text{Ref}1))(?\text{Ref}3))))
  (\text{SEQ}((?\text{Ref}0)(?\text{Ref}1)))(?\text{Ref}3))))
  (\text{SEQ}((?\text{Ref}0)(?\text{Ref}1)))(?\text{Ref}3))))
  \_ST1\_]
```

The listing above shows one transition from state $\text{Start10}$ to $\_ST1\_$. This transition is the BMSC term for the HMSC generated by eliminating the transitions sequentially (from the first connection detected by the algorithm to the last). The complexity of the term further strengthens our proposal for using tools for the conversion of interaction specifications to implementation.
Chapter VIII

Case Study

VIII.A Central Locking System

In order to better understand the need and usage of the tool that we have proposed - M2Code, this chapter walks through the modeling of a problem using our tool. We consider a case study from the automotive domain - Central Locking System (CLS) for an automobile. We first describe the high level requirements of the system, then break it down into key abstractions or components followed by detailed description of the requirements using MSCs. Next, we use M2Code and input these MSCs into it and use the tool to convert these MSCs into state machines and finally simulate the generated system on AutoFocus.

VIII.B High Level Requirements

Due to the increase in use of microcontrollers in automobiles, a lot of the functions that were traditionally manually operated are now being automated. The CLS is one such function. The high-level requirements of such a system are described below:

1. The user of the automobile must be provided with an electronic key to access the automobile.
2. The user must be able to look the doors of the automobile using the electronic key.

3. The automobile must provide visual indication when the doors are locked by blinking the indicator lights.

4. Once the doors are locked, the security monitoring device must be armed so that intrusion attempts can be monitored.

5. The user must be able to unlock the doors of the automobile using the electronic key.

6. The automobile must authenticate the user before the doors get unlocked.

7. The automobile must provide visual indication when the doors are unlocked by blinking the indicator lights.

8. The automobile must log all the unlock requests.

VIII.C Components

Based on the high level requirements described in Section VIII.B, we breakdown the system into the following key abstractions or components.

VIII.C.1 Key Fob (KF)

The Key Fob (KF) represents the electronic key provided to the user. In the requirement section in VIII.B only two types of functions are expected by the electronic key - locking and unlocking. This functionality could be presented to the user as two buttons on the electronic key. The KF takes the physical input from the user and send/receives messages to/from the controller.
VIII.C.2 Control (CONTROL)

As its name suggests, this component is the main controller of the CLS. It receives all the input from the KF, processes it, dispatches it to the appropriate component and finally passes the results back to the KF.

VIII.C.3 Lock Manager (LM)

Different automobiles have different number and configuration of locks. For example, sedans have four door locks and one trunk lock. In order to abstract away the details and configuration of the locks, we define the component Lock Manager (LM) that manages all the locks in the automobile. It receives commands to either lock or unlock from the controller, upon receipt of which, it performs the requested operation on the lock(s) and sends an acknowledgement on the outcome of the operation back to the controller.

VIII.C.4 Lighting System (LS)

Different automobiles have different types of lights and indicators to provide feedback to the user. We abstract the specifics of the indicators from the controller using this component Lighting System (LS). This component receives abstract commands from the controller to activate different indicators based on the type of feedback needed.

VIII.C.5 Security Manager (SM)

We define the Security Manager (SM) as a component that handles the security device in the automobile to protect against intrusion. When the automobile is locked, this component would receive a message to arm the security device.
VIII.C.6 Database (DB)

This component Database (DB), is used to log all the unlock requests made from the electronic key. Due to memory limitations, the actual implementation of the database could vary. Again, abstracting this as a separate component allows us to insulate other components from these differences.

VIII.D Message Sequence Charts

In Section VIII.B we described the various requirements for the CLS and in Section VIII.C we described the key abstractions of the system. In this section, we convert the requirements into interaction specifications in the form of Basic and High Level MSCs as described below. The requirements can be grouped into two sets of scenarios - one for locking and the other for unlocking the automobile.

VIII.D.1 Basic MSCs for locking the automobile

The requirements for locking the automobile are specified in VIII.B.2, VIII.B.3 and VIII.B.4. We convert these requirements into two Basic MSCs - one that describes the locking of the doors as shown in VIII.1a and the other that describes the activation of the security device as shown in VIII.1b. Both these interaction scenarios start when the user presses the “Lock” key on the key fob. In the first scenario, pressing the “Lock” key causes the key fob to send a \texttt{lock} message to the controller. In response to this message, the controller sends the \texttt{lock} message to the lock manager. Once the lock manager completes locking the doors on the automobile, it sends an \texttt{ok} signal back to controller. On receipt of this acknowledgement, the controller sends a message \texttt{door\_lockd\_sig} to the lighting system to provide visual confirmation to the user that the doors are locked. Finally, the controller sends an \texttt{ok} message back to the key fob indicating the completion of this locking operation. In the second scenario, pressing the “Lock” key causes the key fob to send a \texttt{lock} message to the controller. In response to this message the
controller sends the *arm* message to the security manager after which, it sends an *ok* back to the key fob indicating the completion of this locking operation. Both these scenarios occur in response to the same *lck* message sent from the key fob to the controller and hence can be composed into one MSC using the JOIN operator. This is described in detail in Section VIII.D.3.

### VIII.D.2 Basic MSCs for unlocking the automobile

The requirements for unlocking the automobile are specified in VIII.B.5, VIII.B.6, VIII.B.7 and VIII.B.8. We convert these requirements into two Basic MSCs - one that describes the unlocking of the doors as shown in VIII.2a and the other that describes the authentication of the user and logging of the user access in the database as shown in VIII.2b. Both these interaction scenarios start when the user presses the “Unlock” key on the key fob. In the first scenario, pressing the “Unlock” key causes the key fob to send an *unlk* message to the controller. In response to this message, the controller sends the *unlk* message to the lock manager. Once the lock manager completes unlocking the doors on the automobile, it sends an *ok* signal back to controller. On receipt of this acknowledgement, the
controller sends a message `door_unlkd_sig` to the lighting system to provide visual confirmation to the user that the doors are unlocked. Finally, the controller sends an `ok` message back to the key fob indicating the completion of this unlocking operation. In the second scenario, pressing the “Unlock” key causes the key fob to send an `unlock` message to the controller. In response to this message the controller sends a `handle_id` message to the security manager to validate the user. The security manager queries the key fob to find out its identifier by sending it the `get_id` message. The key fob provides its unique identifier to the security manager using the `id` message, after which, it sends an `ok` back to the key fob indicating the completion of this locking operation. Both these scenarios occur in response to the same `lock` message sent from the key fob to the controller and hence can be composed into one MSC using the JOIN operator. This is described in detail in Section VIII.D.3.

Figure VIII.2: Unlocking MSCs UNLK-1 (a) and UNLK-2 (b) for the Central Locking System

VIII.D.3 High Level MSC for the Central Locking System

In order to combine all the Basic MSCs described above, we describe a High Level MSC that composes all these MSCs as shown in Figure VIII.3. Al-
though the Basic MSCs for describing the locking operation are described using two interactions LCK-1 and LCK-2, the locking of the automobile is an interaction specification that is the combination of these two MSCs. However, since both these MSCs describe the interactions based on the exchange of a common messages - \textit{lck} from the KF to CONTROL and \textit{ok} from CONTROL to KF, we cannot combine them by just including their interactions in one MSC. They need to be joined using the JOIN operator. Similarly, the two interactions for the unlocking operation UNLK-1 and UNLK-2 need to be joined as they share a common set of messages - \textit{unlk} and \textit{ok}. The interaction specification can start off by either locking the automobile or unlocking it. This is shown by the two sequential connectors from the START operator in the diagram - one connecting the START to the JOIN of the locking MSCs and the other connecting it to the JOIN of the unlocking MSCs. Additionally, the user can infinitely keep locking and unlocking the automobile. This is shown by a sequential connection from the JOIN of the locking MSCs to the JOIN of the unlocking MSCs and another one from the JOIN of the unlocking MSCs to the JOIN of the locking MSCs. Since these scenarios can be executed infinitely, there is no END operator defined for this diagram.

Figure VIII.3: High Level MSC for the Central Locking System
VIII.E Modeling the Central Locking System using M2Code

Now, we describe the usage of our tool to model the Central Locking System described above. We first start M2Code by invoking its executable. This launches the Microsoft Visio graphical editor. Using the File → New option, we create a new diagram based on the M2Code template. This initializes the model and brings up the index page as the first diagram and the M2Code menu options as shown in Figure VIII.4.

![Figure VIII.4: Modeling CLS using M2Code](image)

Next, we create the four Basic MSCs (LCK-1, LCK-2, UNLK-1 and UNLK-2) and one High Level MSC (CLS-1) using the M2Code → New MSC menu option on the Visio user interface. The individual MSCs are drawn by dragging and dropping the MSC elements from the M2Code stencil shown on the right in Figure VIII.4. Once all the diagrams have been drawn, we can invoke the conversion process by using the M2Code → Generate State Machines menu option on
Visio. This causes a set of forms to be displayed that walk the user through the steps of selecting the components to be converted, the diagrams that need to be included in the conversion process and the initial and final states for that component as shown in Figures VIII.5, VIII.6 and VIII.7. At the end of the last step, the Finish button on the form gets enabled, pressing which, causes the conversion process to start and generate the state machines for all the selected components and the SSD for the entire system (if the check box was selected in Step 1).

Figure VIII.5: Invoking the conversion process - Step 1

Figure VIII.6: Invoking the conversion process - Step 2
VIII.F Interpreting the generated diagrams

In the form selection above, we selected all the components and all the diagrams for the generation process. Additionally, we selected INITIAL as the initial and final state for all the components except CONTROL and both LCKD and UNLD as the initial and final states for CONTROL. The reasoning behind this is that all the automobile can be infinitely locked and unlocked and that the user can start by either locking the automobile or unlocking it. For each of the generated state machines, we provide a brief explanation so that we can understand the generated implementation and how it correlates to the interaction specifications provided as MSCs.

VIII.F.1 KF State Machine

The state machine for the Key Fob is shown in Figure VIII.8. The diagram shows _js0_ as the initial and final state. From this state there are two outgoing transitions one sending the _lck_ message and the other sending the _unlk_ message, which matches the interaction specification described in the High Level MSC in Figure VIII.3. Finally, both these transitions reach state _js1_, after which the KF waits for the receipt of the _ok_ message, after which, it transitions back to
the \_js0\_ state waiting for key presses from the user.

\begin{center}
\includegraphics[width=0.5\textwidth]{fig8}
\end{center}

\textbf{Figure VIII.8: Generated State Machine for the Key Fob}

\section*{VIII.F.2 \ LM State Machine}

The state machine for the Lock Manager is shown in Figure VIII.9. The diagram shows \_js1\_ as the initial and final state. This state has two outgoing transitions one sending the message \texttt{lck} and the other sending the message \texttt{unlk}, both leading to the same state \_js0\_. This matches the interaction specification in Figures VIII.1a and VIII.2a, where it receives either a \texttt{lck} or the \texttt{unlk} message from the CONTROL component. In either case, it responds by sending the \texttt{ok} message back to the CONTROL component. This is shown in the state machine by the transition from \_js0\_ back to \_js1\_ with the transmission of message \texttt{ok}. 

VIII.F.3  LS State Machine

The state machine for the Lighting System is shown in Figure VIII.10. The diagram shows \_js0\_ as the initial and final state. There are two transitions from this state back to itself, one on the receipt of the message \textit{door\_lckd\_signal} and the other on the receipt of the message \textit{door\_unld\_signal}. This matches the interaction specifications described in Figures VIII.1a and VIII.2a, where the LS component receives these messages from the CONTROL component.
VIII.F.4 SM State Machine

The state machine for the Security Manager is shown in Figure VIII.11. The diagram shows \_js1\_ as the initial and final state. There are two outgoing transitions from this state. The first transition occurs with the receipt of the *arm* message causing a transition back to the same state. This first transition back to \_js1\_ matches the interaction scenario described in VIII.1b, where it receives this message from the CONTROL component asking it to activate the security system after the automobile is locked. The second transition occurs due to the receipt of the *handle_id* message causing a set of transitions that finally lead back to this state \_js1\_. This also matches the interaction scenario specified in VIII.2b where it receives this message from the CONTROL component when the user attempts to unlock the automobile.

![Figure VIII.11: Generated State Machine for the Security Manager](image_url)
VIII.F.5 DB State Machine

The state machine for the Database is shown in Figure VIII.12. This shows only one state _js0_ as the initial and final state and only one transition back to the same state on the receipt of the _id_ message. This matches the interaction specification shown in Figure VIII.2b, where the Database component receives the identification of the user from the security manager in order to log the user access in the database.

![Figure VIII.12: Generated State Machine for the Database](image)

VIII.F.6 CONTROL State Machine

The state machine for the Control component is shown in Figure VIII.13. The generated diagram is fairly complicated and hence further strengthens our proposal that it is desirable for the state machines to be automatically generated by a tool from the interaction specifications rather than manually drawing it. The diagram shows _js6_ as the initial and final state. From this state, there are two outgoing transitions one with the receipt of the _lck_ message and the other with the receipt of the _unlk_ message. Transition of both of these messages leads to a set of interleaved transitions that are generated due to the JOIN operators defined in the High Level MSC in Figure VIII.3.

After the interleaved transitions as a consequence of the _unlk_ message
Figure VIII.13: Generated State Machine for Control
complete, a transition to state \_js10\_ occurs, at which point there is a choice to either end the state machine or take a transition to the set of locking related messages with the receipt of the \textit{lck} message. The locking related messages exhibit similar behavior at state \_js0\_, where the state machine can end or take a transition to the set of unlocking related messages with the receipt of the \textit{unlk} message. This matches the interaction scenarios described in Figures VIII.1, VIII.2 and VIII.3.

### VIII.F.7 CLS System Structure Diagram

![CLS System Structure Diagram](image)

Figure VIII.14: Generated System Structure Diagram for the CLS

The generated System Structure Diagram for the Central Locking System is shown in Figure VIII.14. The diagram shows all the components in the system KF, CONTROL, LM, LS, SM and DB as rectangles and arrows between them to represent the channels of communication between them. A channel of communication is shown between two components only if they exchange at least one message. Additionally, messages are sent and received between components in different channels. For example, components CONTROL and LM communicate over channels CL1 (channel for sending messages from CONTROL to LM) and LC2 (channel for sending messages from LM to CONTROL). We can verify that this diagram is generated correctly by counting the different components in Figures
VIII.1 and VIII.2 and the different messages exchanged between them.

VIII.G  Import and Simulation in AutoFocus

After all the state machines are generated by M2Code using the steps described above for the Central Locking System case study, we describe the steps below to import the generated model into AutoFocus and simulate the model using Extended Event Traces (EETs).

Since M2Code converts the generated model into the QML format, it can be directly imported into the tool as shown in Figure VIII.15a. Once the model is imported into the tool, it shows up in the list of projects as shown in Figure VIII.15b. In order to visually compare that the model import occurred successfully, we can verify that a couple of the diagrams imported correctly. The SSD, after it is imported into AutoFocus is shown in Figure VIII.16a and the STD for the
SM component, after it is imported into AutoFocus is shown in Figure VIII.16b. Both these diagrams are identical to the diagrams shown in Visio by M2Code as shown in Figure VIII.14 and VIII.11. Note, however, that the layout is slightly different. This is primarily due to the different forms of layout engines used by the graphical editors of the two tools. Finally, we simulate the imported model in AutoFocus and display the simulation result using EETs. A small section of the simulation result is shown in Figure VIII.17.
Figure VIII.17: CLS Simulation in AutoFocus
Chapter IX

Evaluation

In Chapter I.A, we introduced the various problems faced by engineers in the design and development of complex distributed systems and proposed a solution to alleviate some of these issues and in Chapter III, we described the requirements for a tool that could address these. In the subsequent chapters, we described our solution, its design, architecture and usage for a case study from the automotive domain. In this chapter, we would like to evaluate our solution by looking back at the requirements and ensuring that we were able to meet the goals that we wanted to achieve. Additionally, we look at two modeling tools to compare and contrast against our tool and analyze some of the approaches and design decisions that were taken by their designers and how we could absorb some of these concepts into our tool in order to improve it further.

IX.A M2Code

We have implemented a tool that allows a user to describe the system under design in terms of interactions scenarios in the form of MSCs and provide the ability to automatically generate the implementation details as state machines. Since the tool framework is implemented as a plug-in into Visio, it is able to utilize all the drawing capabilities of the tool and the ability to store and load diagrams from persistent storage.
<table>
<thead>
<tr>
<th>NO.</th>
<th>REQUIREMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The tool must provide a graphical canvas interface for the creation of Basic and High-Level MSCs.</td>
</tr>
<tr>
<td>2.</td>
<td>The canvas interface must allow the user to add graphical widgets and their interconnections.</td>
</tr>
</tbody>
</table>
| 3.  | The tool must provide graphical widgets in Basic MSCs for describing:  
|     | a. a component and its life cycle on a time-line.  
|     | b. the state of a component at a certain point in time on its time-line.  
|     | c. sending/receiving message between components.  
|     | d. references to other Basic MSCs.  
|     | e. the parallel composition of messages or references to other Basic MSCs.  
|     | f. the join of groups of messages or references to other Basic MSCs.  
|     | g. alternative message exchanges between components.  
|     | h. looping over a group of messages or references to other Basic MSCs.  
|     | i. the preemption of a group of messages or references to other Basic MSCs with a group of messages or references to other Basic MSC on the receipt of one or more messages.  
|     | j. the triggering of a group of messages or references to other Basic MSCs due to the occurrence of a group of messages or references to other Basic MSCs.  |
| 4.  | The tool must provide graphical widgets in High-Level MSCs for describing:  
|     | a. start of the MSC.  
|     | b. end of the MSC.  
|     | c. references to other Basic and High-Level MSCs.  
|     | d. the parallel composition of other Basic or High-Level MSCs.  
|     | e. the join of other Basic or High-Level MSCs.  
|     | f. the interconnections in order to describe the type of composition of the elements as either sequential, preemptive or triggering.  |
| 5.  | The tool must provide an interface to select the components that need to be converted from a list of all the components in the current model. |
| 6.  | The tool must provide an interface to list all the Basic and High-Level MSCs that a particular component appears on and allow the user to select from this list the MSCs that need to be included in the conversion process. |
| 7.  | The tool must provide an interface to list all the state markers that the user as added to a particular component and allow the user to select from this list the initial and final states for the conversion process. |
| 8.  | The tool must be able to construct a visual representation of the generated Statechart using graphical layout information. |
| 9.  | The tool must be able to construct a visual representation of the generated SSD using graphical layout information. |
| 10. | The tool must provide an interface to allow the user to store the Basic and High-Level MSCs on persistent storage. |
| 11. | The tool must provide an interface allow the user to load Basic and High-Level MSCs from persistent storage. |
Table IX.2: Translation Requirements

<table>
<thead>
<tr>
<th>NO.</th>
<th>REQUIREMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The tool must be able to convert Basic and High-level MSCs into state machines on a per component level given the initial and final states for that component, using the algorithms described in [23].</td>
</tr>
<tr>
<td>2.</td>
<td>The tool must be able to minimize the generated state machines.</td>
</tr>
<tr>
<td>3.</td>
<td>The tool must be able to generate graphical layout information for the generated state machine.</td>
</tr>
<tr>
<td>4.</td>
<td>The tool must be able to deduce the channels of communication between different components using the message flow described in the selected MSCs and generate an SSD that represents the selected components and the channels of communication between them.</td>
</tr>
<tr>
<td>5.</td>
<td>The tool must be able to generate graphical layout information for the generated SSD.</td>
</tr>
</tbody>
</table>

We created widgets for describing all the operators in both Basic and High Level MSCs and provided the mechanisms for the user to draw both these types of MSCs by dragging and dropping these widgets from the M2Code Visio stencil. This allows us to meet the User Interface requirements specified in Section III.B.

Additionally, in Chapter VIII.E, we demonstrated the usage of the user interface of M2Code that allows the user to select the different components, diagrams and states as input for the conversion process. After taking all these pieces of information, the tool was able to convert these into state machines, generate layout information for them and display them as diagrams on Visio. These capabilities demonstrate that the tool is able to meet the Translation requirements specified in Section III.C.

After the tool converted MSCs to state machines, it was able to export the generated model into two formats. We demonstrated in Section VIII.G how this model could be imported into AutoFocus and simulations could be performed on it. With this, we were able to satisfy the Import/Export requirements specified in Section III.D. Thus, we were able to meet all the requirements that were listed in Chapter III.

Now, we analyze two other tools and identify the different requirements
Table IX.3: Import Export Requirements

<table>
<thead>
<tr>
<th>NO.</th>
<th>REQUIREMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The tool must be able to store the Basic and High-Level MSCs in a format that can be used by the tool to load these models at a later point in time or as required.</td>
</tr>
<tr>
<td>2.</td>
<td>The tool must be able to export the generated state machines, SSDs and the associated layout information into the Query Markup Language (QML) format supported by AutoFocus.</td>
</tr>
<tr>
<td>3.</td>
<td>The tool must be able to export the generated state machines and SSDs into the Extensible Markup Language (XML) format supported by RTCGen.</td>
</tr>
<tr>
<td>4.</td>
<td>The tool must be able to export the Basic and High-Level MSCs, the state machines and the SSDs into formats that can be imported by other graphical editors such as Microsoft Word, Microsoft PowerPoint.</td>
</tr>
</tbody>
</table>

that they would be able to meet and what functionality they provide over and above the requirements that we set forth as part of our proposed solution. We selected these two tools as they are representative of two other possible approaches to implementing a modeling tool. One of the approaches is to provide a graphical framework with support for plug-ins and allow the tool developer to customize the tool for their domain. GME, described in Section IX.B is a tool that falls into this category. The second approach is to develop a fully featured tool for the domain (software modeling in our case) and enhance it capabilities for that domain. AutoFocus, described in Section IX.C is a tool that falls into this category.

IX.B GME

The Generic Modeling Environment (GME) [36] is a tool that is developed by the Institute for Software Integrated Systems at Vanderbilt University. It is a meta-modeling framework that is itself domain independent but allows users to develop domain specific bindings to the tool. The main theme of their tool design is to address the issue that tools are highly coupled with the domain that they are being used for. For example, Matlab is tool that is commonly used for signal processing and hence is not well suited for software modeling.

This approach focusses on the design of a domain neutral modeling frame-
work, into which the domain specific modeling information can be fed using a modeling language that consists of UML class diagrams and an Object Constraint Language (OCL). Once this information is provided to the tool, it generates the domain specific environment. This domain specific tool environment can now be used for design in that specific domain and additionally manage the constraints for that domain based on the information provided in the meta-model. The tool organizes the model using different abstractions such as Project, Folders, Model, Atom, References, Connections, Aspects, Parts, Sets, Roles, Constraints etc. Using these abstractions, the meta-model can be described hierarchically by decomposing it into smaller models, relationships between the models can be described using connections and the different views of a model can be described using parts and aspects.

In order to use GME for modeling software systems using scenario based requirements and automatic synthesis of system design from these; it would involve the following steps. The first step would be to build a model to represent the domain - BMSCs, HMSCs, SSDs and state machines using class diagrams. Additionally, the semantics of each of these diagrams, their constraints and their behavior would have to be described using the OCL accepted by GME. Once all this information is provided to it, the modeling framework can be built that would allow a user to design a software system. The next step would be to implement all the algorithms for processing the domain model and performing the translations by using its plug-in architecture. These plug-ins would be invoked by the framework when the user chooses to convert the MSCs into state machines. This approach of first building the modeling tool would definitely allow for stricter user interface behavior. But, since the meta modeling tool is designed to be very general to accommodate all domains, it might not be a trivial task to customize it for our needs. Additionally, it would tightly couple our implementation to this solution. Our approach of using a general purpose graphical editor, instead, allows us to primarily focus on the extraction of the information provided by the user from the
graphical editor, build a domain model in the form of BMSCs and HMSCs and translate them into SSDs and state machines rather than focusing on the actual behavior and constraints provided by the editor.

IX.C AutoFocus

AutoFocus [1] is a tool developed by Department of Informatics, Technische Universität München. It is a tool that is designed to specifically address issues related to software modeling. The tool itself is a client/server system wherein the model for the system under design can be stored in a central repository and any number of editor clients can run on user machines. Additionally, it is implemented in Java to make it completely platform independent.

This tool focuses on the design of distributed systems by providing the ability to define it from different aspects - structural, behavioral and interaction oriented. It allows the user to define the structural views of the system in the form of SSDs. These are drawn by using its SSD Editor and describe the different components, their channels of communications and the ports on the components that connect these channels to the components. Components can also be decomposed into subcomponents and each of them can also be defined using SSDs and by establishing hierarchical relationship between them. The behavioral views of the system are described using STDs and drawn using the STD Editor. These diagrams allow the user to define the different states of a component and the transitions that occur between them. Each of these states can themselves be composed of smaller substates, which again allow hierarchical decomposition. STDs help describe the behavior of each of the components and also how they respond to external events. Interaction between components can also be described using EETs by using the EET Editor. This allows the user to describe components on a timeline and show these interactions by sending/receiving messages. The interactions described on a particular component may be handled by subcomponents and hence components in
an EET can be associated with other EETs that describe the interaction between its subcomponents.

Finally, once the diagrams have been input into the tool, the user can perform consistency checks on it and simulate the system. The simulation consists of displaying the current state of the system on each of the diagrams by highlighting it. Additionally, if there are situations where the simulation needs to make a nondeterministic choice, the tool can be configured to either prompt the user for a decision or randomly generate one.

Since AutoFocus is a closed source tool and does not support conversion of MSCs to state machines, it is unable to directly fulfill one of the most important requirements that we specified in Chapter III, namely the translational requirements. However, it supports powerful editors to draw and modify structural, behavioral and interactions specifications between components. These would allow it to fulfill some of the user interface related requirements but not all as these editors are very specific in their behavior and only allow modification of the supported diagram types and not for general purpose editing with operations such as cut-n-paste. M2Code, on the other hand, is able to fulfill the translational requirements and provides the ability to export these generated diagrams into AutoFocus. It does not, however, allow editing of the generated diagrams and neither does it support simulation. Hence, when used in conjunction, M2Code can be used to input the MSCs, generate SSDs and state machines and AutoFocus can be used for editing and simulating these diagrams.
Chapter X

Related Work

X.A Introduction

In the preceding chapters we described the problems involved in the design of complex systems and how we address this using our tool - M2Code, which converts behavioral requirements in the form of MSCs into state machines. In this chapter, we describe four other approaches taken by different research groups.

X.B Approaches

In the following sections, we describe each of the approaches by first describing a brief summary, followed by an overview of the methodology, tool and algorithm and finally provide a comparison between their approach and ours.

X.B.1 Labeled Transition System Analyzer - LTSA

LTSA is a tool designed to take MSCs as input and synthesize them into an Labeled Transition System (LTS) model. The tool can then perform model checking on the synthesized LTS for properties such as deadlocks, safety and progress. [30] describes their motivation, approach and algorithms and [31] provides some insight into the implementation and usage of their tool.

They describe the motivation and the need for such a tool by listing
the benefits of using scenarios for system specification as they are easy to use and comprehend during requirements elicitation. However, since scenarios are drawn by multiple stakeholders, the specification may be partial or the individual scenarios may be disconnected. Their approach consists of accepting user input in the form of BMSCs for describing each of the scenarios and HMSCs for describing the composition of these scenarios. Additionally, these BMSCs can also contain state information on either a component level (where they are placed on certain points on the component timeline of a particular BMSC) or at a system level (where they are placed at a certain point on all the components of a particular BMSC).

The tool is designed using a plug-in model and implemented in Java. It has a user interface to allow the input of MSCs. Their synthesis algorithm first builds a Finite Sequential Process (FSP) for on a per-component basis. This is accomplished by adding labeled states at the top and bottom of each component on each BMSC. Then, using the HMSC, the algorithm constructs a relation using the start/end states on each of the MSCs. This relation is used to describe how each of the MSCs need to be combined together. After this, the algorithm iterates through each of the component timelines on the BMSCs and splits them into segments consisting of a labeled start and end state with one or more message exchanges between them. These are then combined using the relationship obtained from the HMSC. Finally, the result of the above steps is used to construct FSPs which consist of a process name (label of the first state) and the sequence of events as the behavior for that process. The FSP for the entire system is obtained by the parallel composition of the FSP of all the components. The tool then uses this FSP as input and converts it into an LTS, which can then be run through the model checker to check for properties such as deadlocks, safety and progress.

Their approach is a little similar to ours in that they accept both BM-SCs and HMSCs as input and utilize user provided state information to aid the conversion process. However, in their approach, they do not provide support for
BMSC operators such as REF, JOIN, PAR, PREEMPT and TRIGGER. In the
HMBC also, they only provide support for alternatives by using the outgoing
arrow from one BMSC going to more than one BMSC or HMSC. Additionally, their
approach assumes the exchange of messages between components as being syn-
chronous. They do not support generation of code from the synthesized models.
Their approach, on the other hand provides support for inputting both positive
and negative scenarios in the form of BMSCs, which can then be verified during the
model checking phase. Additionally, their BMSCs can have both component-level
and system-level states, allowing for more user level hints on the composition of
the BMSCs.

X.B.2 Play-Engine

Play-Engine is a tool that provides a graphical interface to capture sce-
narios using play-in that are then converted by the tool into Live Sequence Charts
(LSCs). Using play-out, these LSCs can be directly executed and analyzed by the
tools’ verification modules. If the system is synthesizable, it also generates state-
charts as described in the UML specification, thereby allowing their execution and
further analysis on UML case tools such as Rhapsody. [15] describes this tool, the
underlying concepts in their approach, the algorithm and a case study.

In this approach, they use LSCs, which are extensions to MSCs, to de-
scribe scenarios. Their notation allows to either describe behaviors that may hap-
pen (existential) or the ones that must happen (universal) or scenarios that must
not happen. Universal charts have semantics similar to the usage of the TRIG-
GER operator described in our MSC terminology. They have the concept of a
prechart, which are scenarios that are enclosed by a dashed hexagonal shape and
placed in the beginning of the LSC. These scenarios in the prechart, if executed,
cause the execution of the remainder of the scenarios described in the LSC. In
order to make it easier for the user, they allow the scenarios to be specified using
the play-in facility of their tool. With this, the user is provided with the graphical
representation of the system and he/she interacts with the GUI to describe the required behavior. The Play-Engine constructs the LSC in the background while the user is describing these scenarios. Additionally, their tool allows importing and exporting UML models from and to Rhapsody. Due to this approach, their internal model uses the abstractions from UML - classes, objects, methods and properties to describe the system under design.

During their analysis, they found that the synthesis of statecharts from LSCs is much harder than from MSCs. They handle this by requiring the user to provide detailed scenarios and knowledge of the underlying design to assist the synthesis algorithm. Their algorithm involves encoding the play-out as a transition system with one process for each object and a set of charts corresponding to each of the LSCs. This transition system considers state at a system level rather than at the object level wherein each state defines the set of charts that are active, the set of messages being sent to every object, the set of messages being received by each object and the location of each object in each of the charts. They apply model checking to this encoding to prove that the play-out will not get into a deadlock and will satisfy all the requirements of the system and if so, they proceed with the synthesis. The synthesis involves creating a statechart for each object from the above model.

Their approach is quite different from ours. They use LSCs for describing scenarios, while we use MSCs. LSCs seem to only support semantics similar to the TRIGGER operator, which allows the user to describe eventual behavior. However, it does not allow other forms of message composition such as ALT, PAR, JOIN etc. Additionally, it does not seem to support semantics for describing the composition of scenarios, for example via HMSCs. This would make it harder to describe the relationships between the different scenarios. Their tool, however provides a convenient way to describe scenarios using play-in, in a simulation-type environment. It is also integrated with a powerful and widely used UML tool - Rhapsody, which allows them to import/export their models and utilize the differ-
ent features supported by that tool. So, although they don’t directly support code
generation, they could probably rely on this feature from Rhapsody. Their tool
also has the benefit that it first runs through a model checker before attempting
to synthesize the system. However, as they have described in their analysis, their
approach is not complete and may not synthesize the system at all in some cases.
Additionally, since their model checking algorithm is applied on the entire system
space, the computation complexity is very high and hence does not scale very well.

X.B.3 MSC Editor, Simulator and Analyzer - MESA

MESA is a toolset that provides graphical interfaces to input MSCs as
described in the Z.120 MSC specification and implements algorithms for analyzing
these. The tool then converts these MSCs into a Real-Time Object Oriented
Model - ROOM, which can then be analyzed in another commercially available
tool called ObjeCTime (now part of Rational Rose RealTime [32]). The generated
ROOM models consist of both structural models and behavioral models. This
toolset, the algorithm and a detailed case study are described in [24].

In this approach, the user describes scenarios in the form of BMSCs and
HMSCs and the tool converts this into structural and behavioral ROOM models.
The structural diagrams in ROOM consist of components, shown as rectangles
and referred to as actors. These actors can be hierarchically decomposed further
into other actors. They can communicate via ports by sending messages asyn-
chronously. Additionally, all the actors in the system are enclosed within one over-
all actor referred to as the system actor. The behavioral diagrams, referred to as
ROOMcharts, consist of finite state machines for each of the actors and are similar
in semantics to the statecharts defined in the UML specification. These diagrams
can also be composed hierarchically and show state transitions with the sent/re-
ceived messages occur on the ports defined in the structural diagrams. Transitions
in ROOMcharts are triggered only by receive events. Each of the actors except for
the system actor, is associated with at least one ROOMchart.
Their algorithm consists of two phases - one for generating the structure and the other for generating behavioral diagrams. The algorithm for structural diagram generation involves identifying the processes in all the BMSCs and creation of one actor for each of these processes and one overall system actor. The protocols between the actors is determined by finding the relationships between the processes in the BMSCs and a carrying out a depth-first search of the HMSC for the relationships caused due to composition of the BMSCs. They support two different algorithms for synthesis of the behavioral diagrams - Maximum Traceability and Maximum Progress. The Maximum Traceability algorithm tries to generate ROOMcharts that maintain the same structure as the input MSCs. For all actors but the system actor, the algorithm creates top-level states to represent each of the nodes in the HMSC, which are then connected by transitions to represent each of the edges in the HMSC. Next, a ROOMchart is generated for each of these top-level states by converting the message exchanges between actors into state transitions. With this algorithm, each actor has one top-level ROOMchart (synthesized from the HMSC) and one ROOMchart synthesized from each of the BMSCs. Note that if an actor is not present in a particular BMSC, a ROOMchart is still created and consists of one state and an empty transition.

Maximum Progress algorithm, on the other hand, generates only one ROOMchart per actor (excluding system actor). Each of these ROOMcharts are generated by first building a message graph per actor by inspecting each BMSC and using the structure in the HMSC for inter BMSC transitions. This message graph is then used to find traces consisting of states and transitions. These traces are then converted into a ROOMchart.

This approach is similar to our approach in some aspects. They also use both BMSCs and HMSCs to describe behavioral requirements and their ROOM structural diagrams are similar to the SSDs and their behavioral diagrams (ROOM-charts) are similar to the state machines that we generate. However, their BMSCs and HMSCs don’t support other compositional operators such as PAR, JOIN,
LOOP etc. They consider composition of BMSCs using HMSCs to be mutually exclusive, thereby indicating that a component can be active in only one BMSC at any point in time and BMSCs can only be composed sequentially. Their generated diagrams, however, allow for hierarchical decomposition, which is currently not supported by M2Code. Since their tool exports the generated models into ROOM, they rely on ObjecTime for features such as simulation and code generation. ObjecTime provides good simulation tools but only generates skeletal c++ code. Our approach, on the other hand, utilizes the simulation capabilities in AutoFocus and complete code generation for RT CORBA middleware using RTCGen.

X.B.4 Minimally Adequate Synthesizer - MAS

MAS is a tool that can be used to generate UML Statechart diagrams from Sequence Diagrams. This tool takes an interactive approach for the conversion algorithms by asking the user to provide answers to membership queries thereby attempting to synthesize diagrams that closely match the users input. [25] describes the overall approach for their tool and [26] describes the algorithm and implementation details.

This approach uses the UML specification for sequence diagrams as a means of providing scenarios into the tool. In addition to the basic semantics of representing a sequence of message transactions between objects, they also support conditional branching, iteration, recursion, object destruction and state labels on objects. The tool takes these sequence diagrams and generates traces in the form of send and receive events at an object level. In order to synthesize statecharts from these traces, the MAS turns to the user to answer questions in the form of membership queries and equivalence queries. Their main reasoning behind using an interactive approach to statechart synthesis is due to the fact that specifications in the form of MSCs or sequence diagrams are usually incomplete as the user may be unable to provide all the details of the system during requirements capture. Statecharts synthesized from such partial specifications may lead to generation
of transitions that were not originally defined in the scenarios, which may not always match what the user actually wants. They attempt to solve this issue by involving the user during the synthesis phase. The core of the algorithm is based on Angluin’s framework of a minimally adequate teacher [8]. In order to reduce the number of queries to the user, the tool maintains the previous user inputs using data structures and uses backtracking if the user changes their decision at a later point in time. These queries accept answers of either *Probably Yes* or *Probably No* and additionally allow the user to defer answering the question to a later point in time. The algorithm also tries to infer answers to unanswered questions, if possible, based on the answers to previous questions or using the information provided in the sequence diagrams in order to reduce queries to the user.

After all the questions have been answered, the tool synthesizes a statechart based on the traces generated from the sequence diagrams. Then it displays this statechart to the user. If the user accepts the result, the algorithm stops, if not, the user is expected to provide counterexamples to guide the algorithm to generate the solution in line with the users expectations. These counterexamples can be either positive or negative. Positive counterexamples are provided to the tool by providing more sequence diagrams, which are then converted into traces again and fed into the algorithm. Negative counterexamples are provided to the tool by editing the generated statechart and removing the unwanted transitions and restarting the synthesis.

This approach differs considerably from our approach even though the inputs and outputs to and from the tool are quite similar - sequence diagrams as input and statecharts as output. Sequence diagrams are similar to MSCs in that they can be used to describe behavioral requirements in the form of scenarios. However, the options for composing multiple diagrams are not as extensive as MSCs. Hence, in this approach, they compose sequence diagrams in a sequential manner and don’t support operators such as PAR, JOIN, PREEMPT, TRIGGER etc and also don’t support hierarchical composition using constructs similar to
HMSCs. Their algorithm also differs from our approach. While they use an interactive approach and ask the user for answers, we take the iterative approach where the user is presented with a result, which can be iteratively refined by modifying the MSCs, if required. Although these approaches are different, the end result of a system designed on both of these tools may be equivalent as the primary focus is to reduce the gaps in the scenario specification by refining it. However, since their tool has to maintain a large amount of data to keep track of the user responses and also to allow backtracking, the tool would need a large amount of storage for complex systems and hence may not scale well.

Table X.1: Comparison of different approaches

<table>
<thead>
<tr>
<th>Scenario Input Format</th>
<th>LTSA</th>
<th>PLAY-ENGINE</th>
<th>MESA</th>
<th>MAS</th>
<th>M2CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario Composition</td>
<td>Supported via BMSC state markers and HMSC connectors (sequential, alternate, looping)</td>
<td>Inherent sequential composition of LSCs due to grouping of scenarios from each diagram</td>
<td>Supported via HMSC connectors (sequential, alternate, looping)</td>
<td>Supported via sequence diagram operators (branching, iteration, state markers) and inherent sequential composition of sequence diagrams due to grouping of scenarios from each diagram</td>
<td>Supported via BMSC state markers, BMSC operators (ALT, PAR, LOOP, JOIN, REF, PREEMPT, TRIGGER), HMSC operators (PAR, JOIN) and HMSC connectors (sequential, alternate, looping, preemptive, triggering)</td>
</tr>
<tr>
<td>Generated Output Format</td>
<td>LTS</td>
<td>Statecharts</td>
<td>ROOM structure, ROOMcharts</td>
<td>Statecharts</td>
<td>SSDs StateMachines</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Not supported</td>
<td>Supported via Rhapsody</td>
<td>Skeletal code through ObjectTime</td>
<td>Not supported</td>
<td>Supported via RTCGen</td>
</tr>
<tr>
<td>Simulation Support</td>
<td>Not supported</td>
<td>Supported natively via play-out</td>
<td>Supported via ObjectTime</td>
<td>Not supported</td>
<td>Supported via AutoFocus</td>
</tr>
<tr>
<td>Model Checking</td>
<td>Supported natively</td>
<td>Supported natively</td>
<td>Supports MSC analysis</td>
<td>Supports MSC analysis</td>
<td>Supported via MSCCheck</td>
</tr>
<tr>
<td>Extensibility/Tool Interactions</td>
<td>Designed with support for plug-ins</td>
<td>Integrates with Rhapsody</td>
<td>Integrates with ObjectTime</td>
<td>Not supported</td>
<td>Integrates with AutoFocus and RTCGen</td>
</tr>
</tbody>
</table>
X.C Comparison

The Table X.1 provides a quick comparison between the different approaches, the inputs used by the tools, the support they provide for composing input scenarios, format of the output and the features they support.
Chapter XI

Conclusion

We have designed and implemented a tool for automatically generating state machines and SSDs from interaction specifications provided as Basic and High Level MSCs. During the implementation of the tool, we realized that various features could be added on to the tool over and beyond the set of requirements that we identified in Chapter III. Below, we provide a brief description of the future work that could be carried out on the tool in order to make it more extensible and enhance its feature set and finally describe a summary of the thesis.

XI.A Future Work

The future work on this tool can be categorized into the following areas - feature enhancements, refactoring of the tool to increase decoupling between the different modules, migration to a different and more extensible plug-in host - Eclipse and finally new extensions that can be integrated into the tool itself.

XI.A.1 Enhancements

The current implementation of M2Code satisfies all the requirements described in Chapter III. However, during the process of using it, we realized that we could implement some more features that would further enhance its feature set.
1. The tool currently does not check for MSC semantic violations while the user is drawing the MSC. Since tool is based on a graphical user interface, it is possible for the user to connect different elements of the MSC in ways that are not permissible by the MSC semantics. One example of this would be a message within an ALT element originating from one component in the first compartment and terminating in the second compartment on another component. Currently, such errors are not caught by the tool and hence would only be detected during the analysis of the generated state machines. It would be desirable if the functionality could be added to M2Code to detect these types of violations while the user is inputting the MSCs into the tool.

2. M2Code provides support for sending messages between components, which is the common form of asynchronous communication. However, it is sometimes useful to switch thread contexts by sending message to self. Support for this type of message is not currently available in the tool.

3. For compartment oriented MSC elements such as ALT, PAR and JOIN, the MSC semantics allows presence of multiple compartments. For example, the ALT, which typically represents an if-then-else block, can be described by two compartments - one for the if-then block and the other for the else block. However, if there are more possibilities, using say a switch-case, then we would need more than two compartments. M2Code currently only supports two compartments in the Visio stencil. The algorithm, however has support for any number of compartments. We do not consider this as a major limitation of the tool as multiple compartments could be implemented by drawing different MSCs and connecting them using references, similar to the approach of replacing the switch-case with if-then-else if ... blocks. This is a little cumbersome to draw, however. Hence, we consider this as a minor inconvenience instead of a limitation in the implementation.

4. For the state machine generation process, the user is required to input the
initial/final states selectable from the list of all state markers added by the user on the component being converted. It would be desirable to allow the user to add new state markers to this list without having to actually add it to the MSC.

5. During the process of conversion from MSCs to state machines, all the original state markers provided by the user are replaced by internal state labels at the end of the minimization stage. It would be desirable to retain the original state markers provided by user, if possible in order to make it easier for the user to analyze the generated diagrams.

6. M2Code currently passes the converted state machine directly to the minimizer and displays the minimized state machine to the user. It would be useful to view the state machines before minimization.

7. After the conversion process, M2Code presents the state machines and SSDs to the user as diagrams on Visio. Since these are diagrams, the user is free to modify them, say in order to rename some of the state labels, add new state transitions, re-layout the diagram etc. These modifications are currently not tracked by M2Code. If the user asks the state machine for that component to be regenerated, then M2Code will regenerate the entire state machine, thereby overwriting the users modifications. Support for modifying the layout of the generated diagrams and relabeling the elements might be a useful feature. However, in order to add new states and transitions, it would be better if the user actually added these pieces of information as interaction specifications so that the tool would generate the appropriate state transitions.

XI.A.2 Tool Refactoring

In Chapter V, we described the architecture and design of the tool. Although the tool is designed in layers and uses different formats to interact with
other tools, there is some amount of code coupling between the code that reads the MSCs from Visio and the one that generates the final state machines. In order to make the tool more extensible, it would be desirable to refactor and componentize it further. The goal of this refactoring effort would be to break the tool into smaller sub-tools as follows:

1. User Interface: This tool would be used to capture Basic and High-Level MSCs from the user and export these MSCs as XML files. Additionally, it would be able to display any Basic MSC, High-Level MSC, STD and SSD provided to it as input in the form of XML files.

2. High-Level MSC Transformer: This tool would take a High-Level MSC as an XML file and transform it into a Basic MSC and output it as an XML file.

3. MSC Converter: This tool would take one or more Basic MSCs as XML files and convert them into SSD and state machines and output the results as an XML file.

4. Exporter: This tool would take XML files representing the model under design and export them into different forms for import into other tools such as AutoFocus and CodeGen.

The current design of M2Code design is similar to the approach defined above except that there is code coupling between the different components as it is built as one executable application. The suggestion here is to split them into individual tools so that they can be designed, implemented and tested independently of each other. The tool design hence would shift from a layered design to a pipeline based design. The two major advantages of this approach are that any of the tools in the pipeline can be replaced by a better implementation without disturbing the entire implementation and new tools that provide different types of value additions can be inserted in the pipeline, again without disturbing the entire implementation. The major drawback of this approach is that crossing of
application boundaries can cause additional delays due to application startup and teardown and the need to parse XML files instead of dealing with memory objects and APIs. Hence, any future work carried out on refactoring M2Code using this approach would have first to carry out an overall analysis in terms of flexibility and processing time before starting any implementation.

XI.A.3 Plug-in for Eclipse

Eclipse [2] is a widely used Integrated Development Environment (IDE) being developed by Eclipse Foundation, an opensource community. Eclipse is being adopted by the software developer community due to its language independence (i.e. the IDE can be used to develop using any of a wide variety of programming languages) and platform independence (i.e. the IDE is available on a wide variety of hardware platforms and operating systems). Additionally, extensibility is one of the important aspects of its underlying tool architecture. This allows other tool developers to develop their own tools and plug them into Eclipse by deriving from a base set of interfaces. The infrastructure within Eclipse manages the configuration of the plug-in and message exchange via APIs and callbacks.

In Chapter IV we described the rational behind our choice to use Visio as the plug-in host for our tool. Although, Visio provided us with all the necessary support in order to meet our requirements, we now realize that there are some limitations in the Visio based solution.

1. Platform Dependence: Visio, being a Microsoft product, is primarily available for the Microsoft Windows family of operating system (OS). This can be a limitation as there is a substantial research and developer community that use UNIX or Linux as the OS on their development stations. Eclipse, however, as described above is available for a wide variety of OSs.

2. Plug-in Availability: Visio is a tool that is primarily used for diagramming. Hence, it is not well suited for software engineering activities such as code
development, refactoring and reengineering. Eclipse, on the other hand, is
widely used for coding and has plug-ins for these and others such as UML.
This makes an Eclipse based solution a one-stop-shop for design, development
and debugging a solution.

3. Cost: Visio is a commercially available application, sold by Microsoft. This
can also prove to be limitation as each developer would have to install this ap-
lication on their development station apart from installing M2Code, leading
to increased cost of installation. Since Eclipse is an open source application,
the basic IDE and code development plug-ins are available for free for down-
load from their website.

When we started our initial design/implementation of M2Code, we eval-
uated Eclipse but at that time, it was still under a lot of development and was also
quite slow as it implemented in Java and is executed via the Java Virtual Machine
(JVM). Visio, on the other hand is a binary optimized for the X86 platform and
is hence fast and responsive. However, Eclipse has undergone a lot of refinement
to make it much more stable and faster. Additionally, there been substantial ad-
vances in the JVM from Sun in order to make it faster even though the application
code is still executed by the interpreter. These have led to a substantial increase
in the adoption of Eclipse by the developer community. Based on all these, we
feel that a desirable enhancement to M2Code would be to migrate M2Code from
being plugin to Visio to a plugin into Eclipse.

XI.A.4  Extensions

In order to extend some of the current functionality provided by our tool
framework, other tools could be integrated using the import/export scheme used
for other tools such as AutoFocus and RTCGen. Currently, the following tools are
being actively worked on at S3EL, UCSD.

1. MSCCheck is a tool being designed and implemented in S3EL that is used
to verify implementation against a property specification. It uses MSCs for system specification and converts them into global automata. Then, using model checking algorithms, it performs checks on an implementation in the form of Buchi automata against the properties defined in the MSC, which are also translated to automata. Finally, it can present counter examples as MSCs in case there are any violations against these properties during the model checking process. It is designed as a plug-in for Eclipse [2]. Adding this tool as an extension to the current tool would enhance it by adding model checking capabilities. This can be accomplished by coming up with a common XML format for describing MSCs between the two tools.

2. As part of the Service Oriented System Design research work being carried out in S3EL, there are ongoing efforts to describe the system in terms of services rather than components. In this approach, interactions are described in terms of roles and the various components in the system play the different roles specified as per the architecture. For further details about Service Oriented Software Architecture, please refer to [27]. The tool that we have designed can be extended to model Service Oriented Systems by implementing an extension that will export the component level MSCs input by the user as XML files, then applying the role mapping from the service description. These modified MSCs would have to be converted back into XML format by the extension and fed back into the tool so that the conversion to state machines can proceed as before. This functionality, could also be implemented as a filter in the conversion pipeline after refactoring the tool as described in Section XI.A.2.

XI.B Summary

We have designed and implemented the goals we defined in Chapter I. We described the methodology of describing requirements as interaction specifications
and automatically generating state machines based on these. The tool that we have implemented is a proof of concept for the rich set of algorithms envisioned [23]. Additionally, it serves as a base tool for further research in this area. We believe that we have contributed to this field of automation tools for software engineering by defining the requirements for a tool that allows designers to be only concerned about defining the interaction specifications and leave the complex part of generating the implementation to the computer. The case study of the Central Locking System demonstrates the usefulness of the tool and reiterates the need and use for such a tool. We have also evaluated our approach with other research that is being carried out in this area. Finally, we have identified areas that we can improve and enhance the features of the tool framework. We have also published a paper that describes the process and tools for service oriented design in the automotive domain [22].
Bibliography


