INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeib Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600
A SOFTWARE TOOL TO DISPLAY MESSAGE SEQUENCE CHARTS

Xiaoming Tang

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

January 1999
© Xiaoming Tang, 1999
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-43559-8
ABSTRACT

A Software Tool to Display Message Sequence Charts

Xiaoming Tang

Message Sequence Charts (MSC) is a trace language. It is widely used to show sequences of messages interchanged between system components and their environment. It is gaining popularity in software engineering methods for concurrent and real-time systems. MSC has been standardized by the ITU-T (International Telecommunication Union, Telecommunication Standardization Sector of ITU) in Recommendation Z.120 since 1992. A new revised standard Z.120 (MSC96) was approved by ITU-T in 1996. MSC includes two syntactical forms, MSC/PR as a pure textual and MSC/GR as a graphical representation. An MSC in MSC/GR representation can be transformed automatically into a corresponding MSC/PR representation. However, the reverse way is difficult since MSC/PR does not include graphical information such as height, width of symbols and texts. This report presents a software tool to convert a textual description into an internal representation, then to display the graphical representation. The object structure built in this software can be used in the further studying or analyzing of MSC96.
Acknowledgements

A great thanks goes to my supervisor, Dr Gregory Butler, for his great patience and valuable guidance.

I am also grateful to Dr Ferhat Khendek for bringing me to the interesting communication world.

Thanks to Steven Li for his help around school.

Finally, I would like to thank my wife, Tong Hu, for her love that keeps me going.
Table of Contents

1 Introduction .................................................................................................................. 1

2 Background Knowledge ................................................................................................. 5
   2.1 Basic Message Sequence Charts ............................................................................ 6
   2.2 MSC structure concept and high-level Message Sequence Charts ...................... 10
   2.3 Our deviation from standard MSC/PR ................................................................. 16
   2.4 Microsoft Visual C++ and MFC ............................................................................ 18

3 The Software Tool ......................................................................................................... 23
   3.1 Overall design ....................................................................................................... 23
   3.2 Detail of parsing the MSC/PR ............................................................................... 26
   3.3 Details of the object structure for MSC ................................................................. 31
   3.4 Details of drawing MSC ....................................................................................... 31
   3.5 Major classes in the tool ....................................................................................... 32
      3.5.1 Classes used to represent MSC element ....................................................... 32
      3.5.2 Classes used to represent high level MSC ..................................................... 38
      3.5.2 Classes represent the sharp of the MSC elements ........................................ 41

4 Conclusion and Future Works....................................................................................... 43

References ........................................................................................................................ 45

Appendix A The Grammar of MSC'96............................................................................. 47

Appendix B Tokens .......................................................................................................... 51

Appendix C List of C Function ....................................................................................... 53
List of Figures

Figure 1 A MSC example in MSC/GR and MSC/PR........................................... 6
Figure 2 MSC timeout in MSC/GR and MSC/PR ............................................. 9
Figure 3 Ordering between two instances ...................................................... 10
Figure 4 Coregion with generalized ordering ............................................... 11
Figure 5 An inline alt example in MSC/GR and MSC/PR ................................. 12
Figure 6 An inline exception example in MSC/GR and MSC/PR ....................... 14
Figure 7 High level MSC disconnection ....................................................... 15
Figure 8 High level parallel MSC ................................................................. 16
Figure 9 The overview of the tool to display a textual MSC ............................... 24
Figure 10 Relationship of representation classes ......................................... 34
Figure 11 Object model for Figure 3 ................................................................ 36
Chapter 1 Introduction

Message Sequence Charts (MSC) can be viewed as a special trace language that shows the sequence of messages interchanged between communication entities and their environment. The communication entities are modeled as SDL services, processes and blocks [1]. The MSC includes two syntactical forms, MSC/PR and MSC/GR. MSC/PR is a pure textual representation and MSC/GR is a graphical representation [7]. The word MSC sometimes is used to refer to a single Message Sequence Chart. A single Message Sequence Chart is used to document a single system, which may be a sub-system of the system. A Message Sequence Chart is not a description of the complete behavior of the whole system. It usually covers a partial behavior and merely expresses the execution trace [2].

Message Sequence Chart’s (MSC) main applications is in the area of telecommunication systems [4]. However, its usage is not restricted to only telecommunications. MSC is increasing its popularity within the software engineering community [3]. MSC has already been adopted by several software engineering methodologies and tools. For example: MSC can be used in the use case model. In the use case model, the user and server can be viewed as instances in the MSC. The user’s request or action can be viewed as a message sent to the server, and the server’s response can be viewed as a message sent back. The trace of the system can be documented as a MSC. The different scenarios can be represented by the inline construct or by high level Message Sequence Chart [5].

Analysis and documentation of a complex system is a crucial and a non-trivial task. MSC can make this kind of task easy. The main advantage of an MSC is its clear
graphical layout that immediately gives an intuitive understanding of the described
system behavior. Firstly, MSC can be used in the software requirement stage to document
the system [4,6]. It is used to visualize sample behavior of a simulated system
specification [4]. Secondly, MSC is used to guide software design, describe the test cases
and scenarios in the test stage. Furthermore, MSC can be used to express system
properties in the verification stage and in the interface specification. Also, MSC is helpful
in software maintenance and reengineering [4,5,6].

MSC has matured within a very short period of time to become a considerably
powerful and expressive language [6]. At the beginning, MSC was used merely as an
informal and illustrative language. Then it advanced to a formal and descriptive language
[6,7]. The International Telecommunication Union (ITU, former CCITT) approved the
first MSC recommendation Z.120 in 1992 [2]. A formal syntax and semantics for MSC
has been standardized in the Z.120. That increased the usage and popularity of the MSC
language. MSC'92 was influenced very much by the idea coming from Petri Nets with its
composition mechanisms based on conditions [7]. However, the language constructs
defined in MSC'92 are not sufficient to describe an information system comprehensively.
MSC'92 has to be used in combination with other languages like SDL. Thus, the study
group of ITU developed corresponding formal semantics based on process algebra [6].
Then ITU approved the new standardization of MSC, which is MSC'96 [1]. MSC'96
becomes a powerful synthesis of concepts taken from process algebra, Petri Nets, and
from object oriented modeling [6]. The object-oriented community has shown interest in
using MSC. Now, it is even under discussion within the "Unified Method for Object
Oriented Development" [13].
The standardization of MSC by ITU makes it possible to provide tools to support it, to exchange MSCs between different tools and to ease the mapping to and from SDL specifications. Many tools are built for this purpose. One example is ObjectGEODE. ObjectGEODE is a software that user can draw and edit a MSC/GR. The elements of the graph are the elements of MSC. ObjectGEODE can transfer the graphic MSC into the textual MSC [15]. However most of those tools are graphical editors based on MSC/GR. They transform an MSC from MSC/GR representation into a corresponding MSC/PR representation [7]. However, transforming an MSC from MSC/PR representation to MSC/GR representation is not so obvious since MSC/PR does not include graphical information such as height, width, or alignment of symbols and texts. One problem is displaying the MSC in the right position. MSC'96 has a very rich grammar, which supports both instance-oriented and event-oriented representations in MSC/PR. In instance-oriented representation, MSC is ordered by instance. And in event-oriented representations, events are listed in form of a possible execution trace. Event-oriented representation is preferable to describe the global ordering. It is closer to the graphical grammar. It is less ambiguity using event-oriented representations in the PR-GR transformation [7]. Mixture of above two syntaxes is allowed in MSC'96. That makes it even harder to transfer a textual MSC to a graphical MSC.

The MSC/PRs are not only used internally by tools, but also edited by human [7]. The MSC is used frequently in connection with the SDL language. The MSC/PR can be analyzed by SDL tools in the cases of validation, tracing, debugging, simulation and test case specification [1,4,6,14]. Also MSC/PR is easy to be convert to process algebra, and to be modified, adjusted during the analysis. So transformation of MSC/PR to MSC/GR.
is important to visually observe the results of modification. Although there is GIF (Graphic Interchange Format), which is a standardized graphic format, for the exchange between different tools, it is basically display the same graphic under different tools, specially in the web site. It is not easy to transfer the element in the graphic as object. Thus, using MSC/PR to exchange and edit between different tools is easy. What do we need is that there are tools to analysis the syntax and semantic of MSC/PR and represent it in MSC/GR.

This paper presents a new software tool, which transfers a textual representation of MSC (MSC/PR) to a graphic representation (MSC/GR). This software will read the textual representation of MSC, analyze it and build a network of objects of class to represent it, then display them graphically.

Chapter 2 briefly reviews the syntax and semantics of a basic MSC and high-level MSC. It also reviews the Microsoft Visual C++ that the software is based on. Chapter 3 describes the design including assumptions, overview, and detail. Chapter 4 provides conclusions, covers some issues of further work, and describes the experience that I have learned. Appendix A presents the grammar of MSC. Appendix B presents the tokens used in the parser. Appendix C presents the C functions of the project. Appendix D presents the classes of the project.
Chapter 2 Background Knowledge

Message Sequence Charts (MSC) is graphical specification formalism. It focuses on the messages interchanged between communication entities and their environment. As mentioned in the introduction, the use of MSC has evolved from describing and visualizing sample, finite system runs and test cases as basic MSC, to describing the behavior of complex system in a modular and hierarchical fashion as high-level MSC [3]. In the first section of this chapter, we review the specification of basic MSC. In the second section we describe the high level MSC. The third section is about our deviation from standard MSC/PR.

Microsoft Visual C++ is a popular software package for programming under Microsoft Windows in PC environment [8]. It also can be used as a C/C++ compiler for writing C/C++ programs under PC platform. The Microsoft Foundation Class (MFC) library has a set of powerful tools to create applications for Windows [10]. In the section 4, a brief review of MFC is given.
2.1 Basic Message Sequence Charts

Basic MSC describes a partial behavior of a system. It describes the finite executions of the processes in the system [1]. Figure 1 shows an example describing a successful connection case. The left side of the figure is MSC/GR. And the right side is MSC/PR.

![MSC connection diagram]

Figure 1 A MSC example in MSC/GR and MSC/PR

Basic MSC contains all constructs that are essential to specify the message flow. Those constructs are instance, message, environment, gate, condition, action, timer, instance creation, and instance stop [7]. The most important construct is instance [2], which is represented as a vertical line delimited by a start and an end in the Figure 1. An instance of an entity is an object of the entity. An entity can be a process, a block, a system or a service. An entity name may be specified for the instance. In Figure 1, instance "Initiator" is an object of entity "process digit".
In MSC/GR each horizontal or sloping arrow represents a message. A tail of the arrow denotes the event message sending. The head denotes the event message consumption. In the MSC/PR, a message inter-change between two instances is split into two events: the message input denoted by keyword in, and message output denoted by keyword out. A message has a name and may have parameters. A message can be sent to "lost", which means it is not consumed. Or a message can be received from "found", which means it appears from nowhere.

The system environment is represented by the frame symbol of MSC, that is, the boundary of the diagram, in MSC/GR, and by the keyword env in MSC/PR. The environment can send or receive messages. In Figure 1, instance "Initiator" receives message "ICONreq" from the environment and then sends "ICON" to instance "Responder".

A gate represents the interface between MSC and its environment. A message sent to the environment or received from the environment is associated with a gate. A gate name can be implicitly or explicitly given. The explicit gate name will appear on the diagram of MSC/GR. In MSC/PR, all gates are defined in the interface by a gate list.

A condition describes the state of a non-empty set of instances of the MSC. When a condition refers to all the instances in the MSC, it describes the global state of the MSC. Otherwise it describes a non-global state. If it only refers to one instance, it describes the local state of the instance. In Figure 1, condition "Disconnected" is a global state that describes the initial state of the communication. A global condition can be an initial global condition, an intermediate global condition, or a final global condition. In
Figure 1, "connected" is the final condition. A condition is defined by the keyword condition in MSC/PR.

An action describes an internal activity of an instance. In MSC/GR, an action is a box that contains the text of the action. The keyword action is used in MSC/PR.

A timer is used to describe timer request, like time expiration, of an instance. Time expiration is represented though set a timer and a subsequence time out. Time supervision is represented though set a time and a subsequence reset timer. Timer is represented by an hourglass in MSC/GR. The keywords set, reset, and timeout are used in MSC/PR.

Using the instance creation and instance stop to represent a dynamic communication system. In such a system, an instance can appear and disappear during the run time. It can be created by another instance and terminate (stop) itself. The instance creation symbol in the MSC/GR is a dashed arrow from parent instance to the head of the child instance. The instance stop is a cross in the end of the instance axis. Only the instance created by another instance can stop itself. The keyword create and stop are used in MSC/PR. Figure 2 shows action, set timer and reset timer.
The order of one MSC is partial ordering. No ordering of message within environment is assumed. Within each instance, time is running from top to bottom; however no proper time scale is assumed. Also no global time axis is assumed for one MSC. The events along the instance axis are ordered (exception in high-lever MSC). Events of different instances are ordered via message(s) or via generalized ordering mechanism among the orderable events of difference instances. A message must be sent before it is consumed. In Figure 2, event “out ICON” is before “in ICON”, so before “out ICONind”. The orderable events are message event, create instance, timer events, and action. The keyword before and a label are used to specify the order in MSC/PR. In MSC/GR is an arrow connecting two events. Figure 3 shows the generalized ordering mechanism.
2.2 MSC structure concept and high-level Message Sequence Charts

The structural language elements are introduced into MSC to specify more general MSC or refine MSC. Those elements are the generalized time ordering, composition and decomposition of instance, MSC reference and composition of events inside the MSC. Then high level MSC is added to MSC. High level MSC focuses completely on the composition aspects. It acts like a road map, which gives a very good characterization how individual MSC composed into an overall system [6]. The structural language elements are coregion, instance decomposition, inline expression, MSC reference, and High-level MSC.

Coregion is used to indict the non-deterministic period in one instance. It marked with starts (keyword concurrent) and ends (keyword endconcurrent). During this time period, event is unordered expect explicit ordering. The explicit ordering can be the generalized ordering mechanism mentioned above, or the timer which must be set before either reset or timeout. Figure 4 show a coregion with generalized ordering. The dashed instance axis is used to represent coregion in MSC/GR.
In Figure 4, "input ICONreq" is before "out IDISind", "in ICONresp", and "out ICONInd". "out ICONInd" is before "in ICONresp". "out IDISind" is unordered with "out ICONInd" and "in ICONresp".

A MSC instance can be referred by another MSC. This technique is called instance decomposition. It is used to determine the transition between different entities of different levels of abstraction. The referred MSC may have explicit name or take the same name as the instance. In both MSC/PR and MSC/GR, the keyword decomposed is used.

The inline expression describes alternative ways for event execution in the MSC. It is used to hold different situation in the presentation of the system. There are five inline express, alt, opt, par, loop, exc.

The alt inline expression means there are two set events and only one set is executed. In such case, MSC has common part of events and separate part belongs to each case. The keyword alt is used in MSC/PR. See the example in Figure 5 that shows the special situation where "user A" is busy.
In Figure 5, in alternative 1, network is waiting for a message from user A but the message is lost. Then timer T is expired. In alternative 2, network received the busy message and reset time T. The gate "g1" is the inline gate used to define the connection point for the inline expression to outside. There are also timeout, send message to lost and gate interface in the Figure 5.

The **opt inline expression** means the part of event of the system is optional. That means the part of system may never happen. It is same as an **alt inline expression** where the second operand is the empty MSC. The keyword **opt** is used in MSC/PR.
The par inline expression means that all events within the parallel sections are executed. The only restriction is that the event order within each section is preserved. The keyword par is used in MSC/PR.

The loop inline expression is the set of events will be executed at least some times (low boundary) and at most some times (upper boundary). The loop may execute infinite time if the upper boundary is infinitive. It may also not be executed at all if the lower boundary is greater than upper boundary. The keyword loop is used in MSC/PR.

The exc inline expression means a compact way to describe exceptional cases in a MSC. The mean of the operator is that either the events inside the exc inline expression are executed and then the MSC is finished, or the events following the exc inline expression are executed. It likes the alt inline expression, which the second part is the entire part of MSC after the exc inline expression. The keyword exc is used in MSC/PR. There is an example in Figure 6.

Figure 6 shows that the system connection is either failure or success. The connection is failed if the response is not coming within time expiration, then the system will remain disconnected. The connection is success if the response is coming in time, then system is connected.
The MSC reference is used to refer other MSC in the same document. It may also have the mapping from the current MSC element to the referred MSC element. Such element can be messages, instances, gates, and MSCs. MSC reference is done by a point to the reference text in MSC/GR and by the keyword reference in MSC/PR.

MSC'96 defines all above structure concepts. In addition to that, it also defines high level MSC (HMSC). The elements of a high level MSC can be MSC (basic MSC or high level MSC), conditions, MSC reference, and connecting point of MSCs. The whole system is divided into objects and each object is a MSC. HMSC explains how the whole system is constructed and how is the overall control. HMSC can also define the alternative execution path or a parallel subsystem. The subsystem may also be described by HMSC.

Often the word MSC refers to high level MSC.
Figure 7 shows a system started from disconnected state and tries to connect. Due to either request message is lost or time is out, the system goes back to disconnected state. MSC “message_lost” is that system receives a connection message from environment and lost inside system. MSC “time_out” is that the request is send out to environment after getting connection message, but time is out before getting response from environment. MSC “disconnection” is that a disconnect indicator is sent to environment. Those basic MSC is not drawn in the figure 7.

```
msc disconnection;
expr L1:
  L1: condition disconnected
      seq (L2 alt L3);
  L2: message_lost seq (L4);
  L3 : time_out seq (L4);
  L4: disconenction seq (L1);
endmsc;
```

Figure 7 High level MSC disconnection

Figure 8 shows parallel composition. The system receives a connection request from “Initiator” and a disconnection request from the ‘Responder’ at a same time. Two requests are parallel. The basic MSCs CR and DR are not showed in the figure.
2.3 Deviation from standard MSC/PR

In this section, some detail of MSC/PR, which is not mentioned in last two sections, but they are related to the project will be discussed. The concrete textual grammar of MSC is listed in Appendix A.

The grammar defines **MSC document**. A MSC document contains one or several MSCs. The MSC inside the same document is visible to others. A MSC can refer to another MSC in the same document. Usually the high level MSC is put with its basic MSC element in the same document. Only one MSC document is allowed for one MSC textual file [2]. The MSC document defined here is different from the text file generated by ObjectGEODE [15].

MSC uses Backus-Naur-Form (BNF) in its grammar. The name of a MSC object is take wider ranger characters in the Z.120. Those characters include notional like ‘#’, or ‘{‘. The grammar used has minor variations from the standard, in that names have a more restrict syntax. Our syntax is defined here:
MSC has three types of comments. First one is started with "/*" and end with "*/", this one is supported by this project. The second one is started with keyword text, but this one only appears in a fix place in the grammar. According to the grammar, it can only appear in the body of basic MSC. The third is started with keyword comment. Z.120 attaches it with the end symbol, ‘;’. This make it can appear in the end of event statement. In this project, I changed to not attaching the end symbol. If the keyword comment is found, all the following characters in the same line will be ignored.

In Z.120, the <identifier> ::= [<< <text> >> ] <name>. In the project, only take the name, as <identifier> ::= <name>.

There are two types of representations, instance-oriented and event-oriented. The MSC/PRs in the Figure 1 to Figure 5 are instance-oriented. The MSC/PR in the Figure 6 is event-oriented. Event-oriented textual representation is new to MSC and more readable than instance-representation. Both representations can be used, but the semantic of mix usage is not very clear [2,7]. User should not mix them in an MSC. Also an instance must declared first before it has event. It can not have event after it is ended.

The gates in the MSC can have order with other gates. A default order out gate is before one event. The default order in gate is before another gate. The display rules are very complicated. Also there are not enough examples can be found. Therefore this project only supports to parser the orderable gate, not to display it.

The grammar of inline expression of MSC makes it possible to have new instance declared inside the inline expression, and also a nesting inline expression. User can
declare a new set of instances inside inline expression and have all events belonged to the new instance. However it is not what the inline expression means and makes cause confusing. Since the structure of MSC reference and high level MSC are powerful, I assume no one uses pervious method. No nesting inline expression is supported although maximum five nesting inline expressions are supported in the parser part. The events inside the inline expression belong to the instances which own the inline expression.

MSC reference can have very powerful combination according to the grammar, but the usage is not clear [7].

2.4 Microsoft Visual C++ and MFC

Microsoft Visual C++ (Visual C++) is a software package based on PC. Visual C++ can be used to compile and build a program, which is written in C/C++. It has a graphic interface for the user to edit their programs, list class hierarchy tree [10]. Visual C++ with its sophisticated application framework, is for professional programmers [8]. The build-in class libraries are tools for user to create applications for windows. The basic library is Microsoft Foundation Class [10].

The window base program processes user input via message from the operation system. MFC simplify programming by hiding the main function and structuring the message-handler process [10]. The AppWizard in MFC is a code generator that creates a working skeleton of a windows application [8]. The base class in MFC is CObject. The general-purpose classes are derived from CObject [10]. Those classes are file manager,
data structure collections, template based classes. **CString** class, **CStringList** class, and template **CList** class are used as basic data structures in this project.

**CObject** is the principal base class of MFC. It serves as the root both for library classes and the classes defined by user. It supports serialization, run-time class information, object diagnostic output and compatibility with collection classes [10].

**CString** represents a variable length string. It is the replacement of C strings and the associated standard C library. It has a constructor which user can pass C style string. It overloads binary operators like compare two CStrings. This class is used to hold the name in the MSC/PR. **CString** can be used as const char* in C.

**CStringList** is a list class, which is a doubly link list for store CStrings. It has following main functionality: adding an element in the head or tail of the list, peeking the list head and tail, removing an element or all elements from the list, accessing the element of the list, inserting an element into the list at any position, iterating through the list, and finding an element.

**CList**, **CList<TYPE, ARG_TYPE>**, is a template version of the list class. It allows user to store any type of data. **TYPE** is type of object stored in the list. **ATG_TYPET** is the type used to reference objects stored in the list. It supports ordered lists of non-unique objects. Access the objects can be done by sequentially or by value.

**CPoint** represents a two-dimensional point with x, y coordinates.

The Graphic Device Interface (GDI) refers to the graphic output functions to windows. GDI separates the window program from physical devices such as screen. All GDI functions appear as the member function of **CDC** class. User can do all drawing through the member functions of a **CDC** object. **CDC** class provides member functions
for device-context operations, working with drawing tools, type-safe graphics device
interface (GDI) object selection, and working with colors and palettes. It also provides
member functions for getting and setting drawing attributes, mapping, working with the
view port, working with the window extent, converting coordinates, working with
regions, clipping, drawing lines, and drawing simple shapes, ellipses, and polygons. CDC
member functions are also provided for drawing text, working with fonts, using printer
escapes, scrolling, etc.

In the AppWizard, CDocument class and CView class are basic classes to
display the draw.

A document represents the unit of data that the user typically stores in. The
CDocument class provides the basic functionality for user-defined document classes.

CDocument class supports standard operations such as creating a document,
loading it, and saving it. The framework manipulates documents using the interface
defined by CDocument.

An application can support more than one type of document. Each type of
document has an associated document template; the document template specifies what
resources are used for that type of document. Each document contains a pointer to its
associated CDocTemplate object.

Application programs interact with a document through the CView object(s)
associated with it. A view renders an image of the document in a frame window and
interprets user input as operations on the document. A document can have multiple views
associated with it. When the user opens a window on a document, the framework creates

20
a view and attaches it to the document. The document template specifies what type of view and frame windows are used to display each type of document.

Documents are part of the framework's standard command routing and consequently receive commands from standard user-interface. A document receives commands forwarded by the active view. If the document doesn't handle a given command, it forwards the command to the document template that manages it.

When a document's data is modified, each of its views must reflect those modifications. CDocument has the UpdateAllViews member function to notify the views of such changes.

A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen, printer, etc. The CVView class provides the basic functionality for user-defined view classes.

A view is a child of a frame window. More than one views can share a frame window. A CDocTemplate object establishes the relationships between a view class, a frame window class, and a document class. When the user opens a new window, the framework constructs a new view and attaches it to the document.

A view can be attached to only one document, but a document can have multiple views attached to it at once. User application can support different types of views for a given document type. For example, a word-processing program might provide both a complete text view of a document and an outline view that shows only the section headings. These different types of views can be placed in separate frame windows or in separate panes of a single frame window.
A view may be responsible for handling several different types of input. A view receives commands forwarded by its frame window. If the view does not handle a given command, it forwards the command to its associated document through a message map.

The view is responsible for displaying and modifying the document's data but not for storing it. The document provides the view with the necessary details about its data. A view access the document's data members directly or by member functions in the document class for the view class to call.

When a document's data changes, the view responsible for the changes typically calls the CDocument: :UpdateAllViews function for the document, which notifies all the other views.
Chapter 3  The Software Tool

This section describes the overview of the software I developed. The software input is a text file, which contains the textual MSC. The software tool has three functions. The first one is parsing a MSC/PR text according to the grammar of MSC'96. The second is building a set of objects of the classes in the MSC/PR. And the other is displaying the shape of those classes in a window.

The requirements of this project are to parse the text format MSC, detect syntax or semantics errors, create the objects to represent the MSC concepts like instance, message and action, and to represent their relationships, and to draw those objects in the window.

3.1 Overall design

The software is divided into three subsystems: parser, MSC structure, and visual classes. Each subsystem represents one of the three functions. The system is not a three layers system. Actually it is like three set of functions or classes dependence on the language. Both C and C++ are used. The overview of the tool is in Figure 9.
Figure 9: The overview of the tool to display a textual MSC

The parser subsystem has a set of C functions for the parser. The parser has three subsets of functions. The first subset functions are used to read string from the text file to the buffers, delete white space, and then get the next token. See appendix B for the list of Token. The second subset functions are parser functions. They uses the token information and production rule of the grammar to check the syntax. Some functions are recursive. The third subset functions are used to build symbol table to check semantics. The symbol table is also used to build the classes. The symbol table is a class. Two symbol tables object pointers are used. One has the name of each MSC and pointer to it. The other one has all the important element of one MSC.
MSC structure subsystem is a collection of classes. It defines the classes and their relationships. It looks like a relational database with entities. The classes are in same level except inline loop class that is a child class of inline class. An object of the class, which represents the element of MSC, represents an object of a MSC. For example: an object of instance class is created for an instance of a MSC. A MSC has a correspond MSC object. Using the construct functions and other element functions, the objects of the classes are built during the parsing. The whole classes relationship diagram looks like a tree. The MSC document is the root node; Basic MSC or HMSC is its child. Class instance is a child of basic MSC, and so on. All the objects can be reached through the object of MSC document object. This object has pointers that pointed to the MSC or HMSC in the MSC document.

The parser functions are the draw force to invoke other functions. The objects of the classes are built during parsing the text file. After a MSC is parsed, all the objects of that MSC are known. The symbol table is used to update some objects.

The draw part is done by calculating the position of each object in the display window. Assign the sharp class to each object. The shape class is assigned to a list in the AppWizard view document. After the document is built, the view class will display those object in the windows. All are done after the parser is finished.

Some restrictions, such as the length of a name, are applied in this project. The whole software is not case sensitivity that means user used lower case is same as the upper case.
3.2 Details of parsing the MSC/PR

The parser sub-system has three sets of functions as mentioned in previous sections. The first set is used for lexical analysis. The second set is used for parser analysis. The last one is a C like class used for symbol analysis. The parser is base on top_down parser method. The input file is only read once to retrieve.

In the parser subsystem, we use some restrictions due to the limitation of the program. The maximum length of a name, L_IDENTIFIER, is 32.

In the lexical analysis, two buffers, each size is 256, are used. Using two buffers is for the lookahead technique. At the beginning of parsing, two blocks of characters are read into the buffer. The function MSC_lx_getchar() is used to get a character; After return the last character in the current buffer, it will switch buffer by pointer switch, and load a new buffer by calling buffer function. Function MSC_lx_lookahead() is used to returning next character to read as a peek. Function MSC_lx_skip_wsp() is used to skip all white space, like space, tab, or comments. MSC_lx_get_token() is to get next token. MSC_lx_get_token() calls MSC_lx_skip_wsp() to skip white space, then gets next character and decides what to do according a global table. The table, which is used for each ASCII character, is an array of structure. The structure contains the function pointer and a token code associated with the character. There are two kind function pointers, which point to function MSC_lx_letter() and MSC_lx_just_return(). Those functions will return the token. MSC_lx_get_token() passes the return token to the calling function. If the character is one of the characters made word, MSC_lx_letter() is called. The word definition is given in section 2.3. In this function, it calls another function to get a stream of characters, which make a word. Then if it is underlined next, call the function to get
another word. Repeat until the string passes the limit of a name or to a character that is
not in the word definition. Then this name string is compared with the keywords, and
returns either the token of the keyword or a token of identifier to indicate the string is a
name. The string is stored for further usage if it is an identifier. MSC_lx_just_return() is
just return the token assigned in the table. For most character, the token returned is an
error token to indicate unexpected character, for example: $, is found. The other is
return the token value, for example: char ';'. The char ';' is a terminal symbol. Each
MSC statement is ended with ';', like in C/C++ language [9,12].

The main assumptions in the lexical analysis is that the input file is a non-empty
text file.

The parser functions are built according to product rule. The parser uses the
recursive descent method [11]. The name of the function starts with MSC_ps_. For
example: rule 11 <msc inst interface> ::= inst <instance list> <end>. The function to
parse it is: MSC_ps_msc_inst_interface() is like

```
MSC_ps_msc_inst_interface(){
    get token inst;
    MSC_ps_instance_list() ;
    MSC_ps_end();
}
```

Another example is for rule 12, <instance list> ::= <instance name> [:<instance
kind>[<instance list>]. The function is like:

```
MSC_ps_instance_list(){
    if ( token == name ) {
        get name;
        if ( token == ':' )
            MSC_ps_instance_kind() ;
        if (token == ';')
            MSC_ps_instance_list();
    }
```
A global variable is used to record the current token. The parser of high level MSC is straightforward. But the parser of basic MSC is difficult and will be discussed in the following parts.

The problem we got is that MSC'96 is not a LL1 grammar [11]. Sometimes the second token is needed to decide which rule should be used. In the event-oriented representation, a name can be the name of instance or the name of event according to the product rule 22 to 30 (in appendix A). The MSC/PR in figure 3 can be rewritten as:

```plaintext
msc TSD;
  inst Initiator, Responder;
  Initiator : instance ;
      in ICONreq from env;
      out ICON to Responder before label1;
  Responder : instance;
      label1 in ICON from Initiator;
  Initiator : out IDISind to env;
        endinstance;
  Responder : in ICONresp from env;
        endinstance;
endmsc;
```

In this example, the identifier Responder is a name of instance, which is the start of <event definition>. The identifier label1 is the name of event, which is start of <orderable event>. Only knowing next token is a name, it is not enough to decide which product rule to be used. A structure, similar to the stack, is used to store token. The push function stores a token to the stack. The retrieve function either returns the store token or gets a new one if the stack is empty. This structure is also used later in the inline expression since inline expression needs two tokens to know its end.
Another problem is that the `<instance name list>` can only have one `instance`. By the rule 22 to 30 and the rule 60 to 65, both following statements are correct:

\[
\text{instance}_\text{name} : \text{condition} \text{condition}_\text{name} \text{shared}; \quad \text{<instance event>}
\]

\[
\text{instance}_\text{name} : \text{condition} \text{condition}_\text{name}; \quad \text{<multi instance event>}
\]

But the following statements are wrong:

\[
\text{instance}_\text{name} : \text{condition} \text{condition}_\text{name} \text{shared}; \quad \text{<instance event>}
\quad \text{alt begins} \text{inline}_\text{name}; \quad \text{<multi instance event>}
\]

\[
\text{instance}_\text{name} : \text{condition} \text{condition}_\text{name}; \quad \text{<multi instance event>}
\quad \text{out message}_1 \text{to env}; \quad \text{<instance event>}
\]

since they mix the product rule of `<instance event list>` and `<multi instance event list>`. In such case, using which product rule can not be decided until the keyword `shared` is found or not found. Then the parser knows the rule for next event statement. A set of functions and a local static variable are used to deal with this problem. The variable is set to be single instance statement at the beginning of rule 22. Then it changes the value if multi instance event statement is found. The above problem is also the reason why the keyword `shared` should be in the statement even a condition is only for an instance.

Some product rules share one function, like rule 76 and rule 82. This is because the above reason and they are similar. In the grammar is the in rule 81 to 86. `<inline expr>` calls back to `<msc body>`. The parser is push the local variable, which is used to distinguish `multi-instance` event list and single instance event list, to another stack and pop out later.

In symbol analysis, a class, `MSC_table`, is used. This table is a hash table using double hash functions. The node of the table is a structure. The structure has name of the element (used as a key), name type (the name is event name, ..), status (like declared,
defined), and a general pointer pointed to the object (like to the instance). The class has following public element functions.

- `search_table()` : search table by the name.
- `insert_table()` : insert a element into the table. If the element is already in table, either ignore the insert or print a error message. The message name can be duplicated. The other names are not.
- `update_table()` : either updates the status or the object pointer by key name.
- `get_ptr()` : return the object pointer.
- `get_status()` : return the status by the name or the position.
- `update_event_link()` : update the pointer of orderable event. Since the after event is only known after all MSC is parsed.

A global table is created for the **MSC document**. It contains the name of the MSCs in the document. For each MSC, a local `MSC_table` object is created. This table contains all the name of **instance**, **message** name, **event** name, etc. defined in the MSC.

In MSC'96, an **instance** is declared by an instance head statement and ended by an instance end statement. Each **instance** must have a header and an end statement. An instance can have event only after it is declared. The other instances can have message statement related to it. An instance can not have any event after it ends. If there is a instance interface in the MSC header, it must contains the same instances as specified in the msc body.

A function `check_inst_in_table()` is used when a instance name is found. It will insert into the table if not exist in the table, or check the status in the table. Another
function is used to check if the instance declared in the head statement is match to the one in interface.

3.3 Details of the object structure for MSC

Building the represent classes is done in the parser function. An instance object is created as soon as it is insert into the local table since all its events have to be attached to the event list. It also makes it possible for other object to point to it. In the basic MSC, only instance name, inline expression name, and MSC reference name may appear before its declaration. In such case, the name is in the message address. Also the instance name may be in the shared list of shared event before it is declared. The type of an instance in the interface, if has one, must be same as its head statement. The instance object is updated when the instance is declared.

The message object is created right after it is first declared. A message from instance to instance must appear in both instances event lists. So it is checked in the second time declaration. Other objects are created in their declaration function and built across several parser functions.

3.4 Details of drawing MSC

Each class is associated a sharp. For example: the sharp of a message is an arrow. After the parser is done, all the objects of the MSC are known. Currently we use the fixed window size in this project. In the basic MSC, the number of instances and the window’s horizontal size decides for the horizontal distance between instances. The vertical size is close to the window’s vertical size. The number of event is counted for each instance. After the largest number of events is known. The largest number and the
window's vertical size decide for the vertical distance between events of an instance. The event horizontal position is aligned to the axis of the instance. For the event across event, the vertical position is adjusted. The draw sharp of each object is added to the view list once.

The document list is used. It is a template list that can store an arbitrary collection of all sharps in any sequence. The draw class defined in the project has the capability to draw itself. Then draw in window is accomplished by stepping through the sharp list.

3.5 Major Classes in the tool

In this sector, we will discuss the major classes used in represent the MSC. There are two set of classes. One set is used to represent the elements of MSC. The other one is used to represent the sharp of the elements.

All classes are derived classes from msc_element class. The class msc_element itself is a derived class from CObject.

3.5.1 Classes used to represent MSC element

The major classes to represent the elements of MSC are:

- msc_element: Base class to represent an element in MSC.
- msc_document: represents the whole document of MSCs.
- msg_seq_chart: represents the basic Message Sequence Chart.
- HMSC_expression: represents the high level MSC.
- instance: represents the instance in a basic MSC.
- msc_gate: represents the gate in a basic MSC.
• `order_inst_event`: represents the orderable event of an instance in basic MSC.

• `message`: represents the message in a basic MSC.

• `msc_condition`: represents the condition in MSC.

• `msc_timer`: represents the timer in basic MSC.

Class `msc_inline`, `msc_reference`, `simple_event`, `msc_action`, and `create_instance_class` are used to represent other elements. The relationships between classes are showed in the figure 10 and explained after the figure.
The msc_element class contains same important attributes for name, sharp, and position. Those attributes are the common parts of all MSC elements. Another good point to have this base class is that an instance can have a list of msc_element. The pointer of the instance msc_element list can be assign to different element, like condition or
orderable event. All the classes are "declare dynamic" [10]. By that, the program calls run-time lib to get run-time information about an object's class. Some virtual functions are defined in msc_element and its deriving classes.

The msc_document class has two lists of pointers. One is a pointer list of basic MSC. This list has pointers pointed to all the basic MSCs in the document. The other list is a list of pointers of high level MSC. This list has pointers pointed to all the high level MSCs. If viewing the relation of elements of MSC as, then msc_document is the root of the tree. A basic MSC or a high level MSC is belongs to one MSC document. There is only one msc_document object for one MSC'96/PR file.

To make things straightforward, I am going to discuss HMSC_expression after I finish all elements of basic MSC.

The msg_seq_chart (basic MSC) class has two pointer lists. One is instance pointer list and the other one is a gate pointer list. Each instance of the MSC has a corresponding pointer in the instance pointer list. An instance is visible inside MSC. The gate pointer list records all the default in gates and default out gates of the MSC. The default order in gate and default order out gate are not represented in this tool since I can not understand them clearly. The class msc_gate will be discussed in the end of basic MSC classes.

The instance class contains the instance kind (process) information defined in the instance head statement. Instance has an important list, inst_event_list. This list is a msc_element pointer list. Since all the MSC elements are the derived class of msc_element, the class of the objects pointed to can be determined in run time. With the advantage of virtual, even no class information is needed. When an instance has an
event, such as condition, the pointer to the event is added to the list. The pointed classes from instance are: order_inst_event, msc_condition, msc_inline, msc_reference, and simple_event. Some event, like condition, can belong to one or more instances. Some event, like the order_inst_event, is only belonged to one instance. An instance can have many events.

The order_inst_event is used to represent the orderable events. It has a pointer pointing to the event. The point is point to a msc_element object, which can be an object of class message, class msc_timer, class msc_action, or class create_instance_class. Order_inst_event also has a list of pointer pointed to the events ordered after it if there is one. For example: the statement

in ICONreq from env before label1, label2

in Figure 4 will created a order_inst_event object. This order_inst_event object has a pointer pointed to message, "in ICONreq from env". The object also has a pointer list that has two pointers pointed to two other objects of order_inst_event. These two objects have name label1 and label2.

Class msc_timer has attributes to record the name and duration of the timer and the action type(set, reset, timeout).

Class msc_action has a CString attribute to hold the action string.

Class create_instance_class has attributes of the created Instance name, parameter list (CString list), and a pointer pointed to the instance.

Class message is another important class. It has a message type. This type records that the message is from what's kind of element to what's kind of element. For example: from env to instance. It has two CString objects to hold message from element name
and send to element name. It also has two pointers pointed to from/to elements. Message
sent to (or received from) environment has one NULL pointers. A pointer is NULL
indicates the side is unknown. Message has a lost type to indicate if the message is lost or
not. A lost can be sender lost or receiver lost. Usually one pointer of the lost message
will be NULL. But a lost message may still have both side information [2]. For example:
sender lost may still have send name. Message has a parameter list. The pointers of
message may point back to instances. Two instances may have two same message
statements and will have same message object through different order_inst_event.

Class condition has a name list since its name can contains several strings. It has
an instance name list, which has all instances that shares the condition. It also has an
instance pointer list pointed to those instances.

The simple_event class is a mixed usage class. It is used to represent the event
that doesn’t need any other attribute, for example: instance stop. It current handler
following events: instance start, instance end, instance stop, coregion start, coregion
end, Inline separate line, and inline end line. we will discuss inline staff later. The
coregion start(or coregion end) tells when the enter (or leave) the coregion.

The inline class actually only contains the inline head information. we haven’t
built a real inline class. A real inline class must have a instance list and the list must be a
subset of the instances that shared the inline expression. All the events inside the inline
expression must be assigned to the instance in the inline instance list not to outside
instance. It is also possible of nesting inline expression. That also make the draw
function very complicated. It is left for further work. Now the inline class have two gate
lists for the gates attach to the inline expression. Two lists are used to distinguish the
gates of different alternatives. An instance list is used to point to the instances that shared the inline expression. All the events inside the inline expression are assigned to the origin instance. The origin instance has the simple events to know when it goes into second alternative and when the inline expression is ended.

The msc_loop_expr is a derived class of msc_inline. It has addition loop boundary information.

The msc_reference class only records the reference string. Further work can be done to have a pointer list to MSCs.

The msc_gate class is used to represent the default input gate, default output gate, inline input gate, inline out gate, actual input gate, and actual output gate. Theordable gate is left for further work. The msc_gate has a type to indicate what kind of gate it is. It has one message pointer since a gate can be implicitly defined by the direction of the message through the gate and by the message name. A gate also can be explicitly defined by a name.

3.5.2 Classes used to represent high level MSC

Now we am going to discuss the classes represent high level MSC.

Class HMSC_expression has name list for its name. It has a HMSC_expression_node pointer list pointed to the node it has.

Class HMSC_expression_node has a pointer pointed to current node, and a list of pointers pointed to the node followed it.

Class HMSC_node has a type to indicate the node type, a name for the referred MSC or condition, or msc_reference. It also has a HMSC_expression pointer list to
point to the sub-expression belongs to the node. That sub-expression will be a parallel
high level MSC expression.

The object model of Figure 3 is showed in Figure 11.
Figure 11 Object model for Figure 3
3.5.3. Classes represent the sharp of the MSC elements

The classes, which are used to represent the sharp of the MSC elements, are quite straightforward. Like the classes representing the MSC element, there is a base class, CElement.

CElement is a derived class from CObject. It has common attributes of the sharps used to display the element of MSC. Those attributes are: color (type COLORREF), EnclosingRectangle (CRect), Name (CTSTR), and Name position point (CPPoint). It has a virtual function draw, which take a CDC parameter.

Class CLine is used to draw a line. It needs the start and end point as its parameters.

Class CDashLine is used to draw a dash line. It also needs start and end point as parameters.

Class CRectangle is used to draw a rectangle. It uses an Enclosing Rectangle to transfer its parameter. So it also only needs two parameter Start point(left up point) and End point(right bottom point).

Class CLineArrow is used to draw an arrow. It uses a start and a end point. Current only vertical or horizontal arrow can be drawn.

Class CDiamRect is used to draw the Diamond Rectangle, which is used to present the condition. It is a collection of several line segments. It takes four parameter: one is used to pass a View point into the function, two is used to pass the drawing start and end point, another one is used to pass the visual unit Vunit.

Class CTimeSetSymb is used to draw the MSC timer set symbol. It is a collection of several line segments. It also takes four parameter: one is used to pass a
View point into the function, two are used to pass the drawing start and end point, another one is used to pass the Vunit.

Class CTimeResetSymb is similar to the class CTimeSetSymb, but it is used to present the MSC timer reset. It takes the same four parameters.

Class CTimeOutSymb is also similar to the class CTimeSetSymb, but it is used to present the MSC timer out. It takes the same four parameters.

Class CCircle is used to draw a circle. It is used to present the HMSC node, and used to draw the message found/lost symbol. It needs circle center point and the radium value as parameter.

Class CMsgFoundSymb is used to draw the MSC message found symbol. It is a collection of a line and a circle. It takes four parameter: one is used to pass a View point into the function, two is used to pass the drawing start and end point, another one is used to pass the Vunit which is used to determinate the radium of the circle.

Class CMsgLostSymb is similar to the class CMsgFoundSymb. It is used to present the MSC diagram of message lost. It also takes the same four parameters.

Class CText is used to present the text in the MSC diagram. It takes two parameter: one is a point that passes the start point of the text, another one is a CString which pass the text.
Chapter 4  Conclusion and Future Work

The main goal of this project is to display Message Sequence Charts. This project is the first step towards a more powerful tool that can display and edit MSC'96 model and be compatible with other existing MSC tools. Another goal is to learn software design and programming under the Windows environment.

The main steps in the tool of this project are parsing MSC/PR, creating the object structure to represent the MSC, and displaying under the Windows environment.

The grammar of MSC'96 is not easy to parse. The C language is used in this part because it is fast and efficient, and there are tools available for compilers [11].

The object structures to represent elements of MSC'96 are done in C++. It takes advantage of the object paradigm, which is supported by C++ [10]. The elements of MSC'96 have object characteristics, which makes it easy to convert MSC to an OO model.

The display part is done under Microsoft Windows because now PCs are widely used, and the windows program became more and more popular. Many software tools are built under Windows, and the Microsoft MFC library is an open library with many programming tools [10].

All the 22 examples listed in the end of the Z.120 standard are tested. The parser correctly identifies all errors in the examples. The object structure is correctly build for all examples. However the display part is not working in all cases now. It is due to the limitation of the time. Further work is needed in this area.

Through the work of the project, I gained a lot of valuable experiences about parsing complex languages, OO design, C++ and MFC. First of all, I now know the
importance of overall design and planning ahead. I learnt to make my design flexible for any future modification. Secondary, Programming under different environments is a quite valuable lesson. Now I clearly know the difference of the input/output method under UNIX and windows: the UNIX uses device driver to control I/O, while the window uses the message. Furthermore, I learnt the resource based program under windows with visual C++, i.e. we can store data in a resource file using a number of established format. Another thing I learnt in MFC is using the dynamic linked library. Now I knew how to use MFC run time type information and how to link the document to its view. In addition, in this project I mixed the C structured design and C++ object oriented design. I used almost all the C++ OO design features: class hierarchies, template, access control, operator overloading, friend class member, polymorphism, etc. All these are very good experiences for me.

The tool I built is far from perfect. Many improvements need to be done, say as the following.

Create a class to fully encapsulate MSC inline. This class may be similar to msg_seq_chart class. Also the class for MSC reference should be able to handle all kinds of references.

The display should be improved to allow screen scrolling, multiple display windows, and to link related diagrams.

The whole software can be improved to be clear and reliable.
References


Appendix A  The Grammar of MSC’96

In the grammar of MSC’96, a terminal symbol is either indicated by not inside <> or it is one of <name> and <character string>. Keywords are in bold. A non-terminal symbol is indicated inside < and >. A production is given for each non-terminal symbol.

The symbol ‘l’ is used for alternative productions; ‘{ and }’ means the element is group together; * is used to indicate the group is for optional and can be further repeated any number of times; * is used to indicate the group must appear once and can be further repeated any number of times; [ and ] means optional. A underlined part stresses a semantic aspect of the symbol. For example: <msc name> means the name is a MSC name. It is same as <name>.

The Backus-Naur-Form of the textual token is not include in this appendix. The important tokens are:

<alphanumeric> ::= <letter> | <decimal digit>
<full stop> ::= .
<underline> ::= _
<word> ::= <alphanumeric> | <full stop> | <alphanumeric> | <full stop> | *
<name> ::= <word> | <underline> <word> | *

Also <character string> means a string inside two apostrophes. For example, 'Take action'.

The start symbol is <msc document>. Rule 1 and 6 are the rules that deviate from the standard for this project.

1 <end> ::= ;
2 <msc document> ::= <msc document head> <msc document body>
3 <msc document head> ::= <document head>
4 <msc document body> ::= {<message sequence chart>} *

5 <document head> ::= mscedocument <msc document name> [related to < sdl reference>] <end>
6 <sdl reference> ::= <sdl document identifier>
7 <identifier> ::= <name>

8 <message sequence chart> ::= msc <msc head> {<msc body> | expr <msc expression>} endmsc <end>
9 <msc head> ::= <msc name> <end> [<msc interface>]
10 <msc interface> ::= [msc inst interface] [msc gate interface]
51 <actual order out gate> ::= <gate name> before <order dest>
52 <order dest> ::= <event name> | { { { <ref identification> } } } via <gate name>
53 <actual order in gate> ::= <gate name>
54 <def order in gate> ::= <gate name> before <order dest>
55 <def order out gate> ::= <gate name>
56 <inline out gate> ::= <def out gate>
[ external out <msg identification> to <input dest>
57 <inline in gate> ::= <def in gate>
[ external in <msg identification> from <output dest>
58 <inline order out gate> ::= <gate name> [external before <order dest>
59 <inline order in gate> ::= <gate name> before <order dest> [ external]
60 <shared condition> ::= <condition identification> <shared>
61 <condition identification> ::= condition <condition name list>
62 <condition name list> ::= <condition name> { <condition name> }*
63 <shared> ::= shared [{ <shared instance list> | all}
64 <shared instance list> ::= <instance name> [ <shared instance list>]
65 <condition> ::= <condition identification>
66 <timer statement> ::= <set> | <reset> | <timeout>
67 <set> ::= set <timer name> [ , <timer instance name> ] [ <duration name>]
68 <reset> ::= reset <timer name> [ , <timer instance name>]
69 <timeout> ::= timeout <timer name> [ , <timer instance name>]
70 <action> ::= action <action character string>
71 <create> ::= create <instance name> [( <parameter list> )]
72 <stop> ::= stop
73 <coregion> ::= concurrent <end> <coevent>* endconcurrent
74 <coevent> ::= <orderable event> <end>
75 <shared inline expr> ::= <shared loop expr> | <shared opt expr> | <shared alt expr> |
<shared par expr> | <shared exc expr>
76 <shared loop expr> ::= loop [ <loop boundary> ] begin <inline expr identification> ] <shared> <end>
[ <inline gate interface> ] <instance event list>
loop end
77 <shared opt expr> ::= opt begin <inline expr identification> ] <shared> <end>
[ <inline gate interface> ] <instance event list>
opt end
78 <shared exc expr> ::= exc begin <inline expr identification> ] <shared> <end>
[ <inline gate interface> ] <instance event list>
exc end
79 <shared alt expr> ::= alt begin <inline expr identification> ] <shared> <end>
[ <inline gate interface> ] <instance event list>
{ alt <end> [ <inline gate interface> ] <instance event list> }*
alt end
80 <shared par expr> ::= par begin <inline expr identification> ] <shared> <end>
[ <inline gate interface> ] <instance event list>
{ par <end> [ <inline gate interface> ] <instance event list> }*
par end
81 <inline expr> ::= <loop expr> | <opt expr> | <alt expr> | <par expr> | <exc expr>
82 <loop expr> ::= loop [ <loop boundary> ] begin <inline expr identification> ] <end>
[ <inline gate interface> ] <msc body>
loop end
83 <opt expr> ::= opt begin <inline expr identification> ] <end>
[ <inline gate interface> ] <msc body>
opt end
84 <exc expr> ::= exc begin <inline expr identification> ] <end>
[<inline gate interface>] <msc body>
ex end
85 <alt expr> ::= alt begin <inline expr identification> ] <end>
[<inline gate interface>] <msc body>
( alt <end> [<inline gate interface>] <msc body> )* alt end
86 <par expr> ::= par begin <inline expr identification> ] <end>
[<inline gate interface>] <msc body>
( par <end> [<inline gate interface>] <msc body> )* par end
87 <loop boundary> ::= '<<' <inf natural> [ , <inf natural> ]>>' 88 <inf natural> ::= inf | <natural name>* 89 <inline expr identification> ::= <inline expr name> 90 <inline gate interface> ::= ( gate <inline gate> <end> )* 91 <inline gate> ::= <inline out gate> | <inline in gate> |
[<inline order out gate> | <inline order in gate>]
92 <shared msc reference> ::= reference [ <msc reference identification> : ] <msc ref expr> <shared> [<reference gate interface>] 93 <msc reference> ::= reference [ <msc reference identification> : ] <msc ref expr> [ <reference gate interface>] 94 <msc reference identification> ::= <msc reference name> 95 <msc ref expr> ::= <msc ref par expr> ( alt <msc ref par expr> )* 96 <msc par expr> ::= <msc ref seq expr> ( par <msc seq par expr> )* 97 <msc seq expr> ::= <msc ref exc expr> ( seq <msc exc par expr> )* 98 <msc ref exc expr> ::= [ exc] <msc ref opt expr> 99 <msc ref opt expr> ::= [ opt] <msc ref loop expr> 100 <msc ref loop expr> ::= [ loop] [ <loop boundary>]
( empty <msc name> [ parameter substitution] ) | (<msc ref expr> )
101 <parameter substitution> ::= subst <substitution list> 102 <substitution list> ::= <substitution> [ , <substitution list> ]
103 <substitution> ::= <replace message> | <replace instance> | <replace msc>
104 <replace message> ::= [ msg] <message name> by <message name>
105 <replace instance> ::= [ inst] <instance name> by <instance name>
106 <replace msc> ::= [ msc] { empty | <msc name> } by { empty | <msc name> }
107 <reference gate interface> ::= ( <end> gate <ref gate> ) *
108 <ref gate> ::= <actual out gate> | <actual in gate> | <actual order out gate> | <actual order in gate>
109 <decomposed> ::= decomposed [ <substructure reference> ]
110 <substructure reference> ::= as <message sequence chart> name
111 <msc expression> ::= <start> <node expression>* 112 <start> ::= <label name> ( alt <label name> )* end
113 <node expression> ::= <label name> : ( <node> seq ( <label name> ( alt <label name> )* ) end ) <end>
114 <node> ::= ( <msc ref expr> )
| empty | <msc name> 115 <par expression> ::= expr <msc expression> endexpr ( par expr <msc expression> endexpr ) *
## Appendix B Tokens

<table>
<thead>
<tr>
<th>Token symbol</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_NONE</td>
<td>Not token, error char</td>
</tr>
<tr>
<td>T_L_B</td>
<td>Left '('</td>
</tr>
<tr>
<td>T_R_B</td>
<td>Right ')'</td>
</tr>
<tr>
<td>T_COMMA</td>
<td>';'</td>
</tr>
<tr>
<td>T_COLON</td>
<td>':'</td>
</tr>
<tr>
<td>T_S_COMMA</td>
<td>';'</td>
</tr>
<tr>
<td>T_APOSTROPHE</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>T_LT</td>
<td>'&lt;'</td>
</tr>
<tr>
<td>T_GT</td>
<td>'&gt;'</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>identifier</td>
</tr>
<tr>
<td>T_ACTION</td>
<td>keyword action</td>
</tr>
<tr>
<td>T_ALL</td>
<td>keyword all</td>
</tr>
<tr>
<td>T_ALT</td>
<td>keyword alt</td>
</tr>
<tr>
<td>T_AS</td>
<td>keyword as</td>
</tr>
<tr>
<td>T_BEFORE</td>
<td>keyword before</td>
</tr>
<tr>
<td>T_BEGIN</td>
<td>keyword begin</td>
</tr>
<tr>
<td>T_BLOCK</td>
<td>keyword block</td>
</tr>
<tr>
<td>T_BY</td>
<td>keyword by</td>
</tr>
<tr>
<td>T_COMMENT</td>
<td>keyword comment</td>
</tr>
<tr>
<td>T_CONCURRENT</td>
<td>keyword concurrent</td>
</tr>
<tr>
<td>T_CONDITION</td>
<td>keyword condition</td>
</tr>
<tr>
<td>T_CONNECT</td>
<td>keyword connect</td>
</tr>
<tr>
<td>T_CREATE</td>
<td>keyword create</td>
</tr>
<tr>
<td>T_DECOMPOSED</td>
<td>keyword decomposed</td>
</tr>
<tr>
<td>T_EMPTY</td>
<td>keyword empty</td>
</tr>
<tr>
<td>T_END</td>
<td>keyword end</td>
</tr>
<tr>
<td>T_ENDCONCURRENT</td>
<td>keyword endconcurrent</td>
</tr>
<tr>
<td>T_ENDEXPR</td>
<td>keyword endexpr</td>
</tr>
<tr>
<td>T_ENDINSTANCE</td>
<td>keyword endinstance</td>
</tr>
<tr>
<td>T_ENDMISC</td>
<td>keyword endmisc</td>
</tr>
<tr>
<td>T_ENV</td>
<td>keyword env</td>
</tr>
<tr>
<td>T_EXC</td>
<td>keyword exc</td>
</tr>
<tr>
<td>T_EXPR</td>
<td>keyword expr</td>
</tr>
<tr>
<td>T_EXTERNAL</td>
<td>keyword external</td>
</tr>
<tr>
<td>T_FOUND</td>
<td>keyword found</td>
</tr>
<tr>
<td>T_FROM</td>
<td>keyword from</td>
</tr>
<tr>
<td>T_GATE</td>
<td>keyword gate</td>
</tr>
<tr>
<td>T_IN</td>
<td>keyword in</td>
</tr>
<tr>
<td>T_INF</td>
<td>keyword inf</td>
</tr>
<tr>
<td>T_INLINE</td>
<td>keyword inline</td>
</tr>
<tr>
<td>T_INST</td>
<td>keyword inst</td>
</tr>
<tr>
<td>Identifier</td>
<td>Keyword</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>T_INSTANCE</td>
<td>instance</td>
</tr>
<tr>
<td>T_LOOP</td>
<td>loop</td>
</tr>
<tr>
<td>T_LOST</td>
<td>lost</td>
</tr>
<tr>
<td>T_MSC</td>
<td>msc</td>
</tr>
<tr>
<td>T_MSCDOCUMENT</td>
<td>mscdocument</td>
</tr>
<tr>
<td>T_MSG</td>
<td>msg</td>
</tr>
<tr>
<td>T_OPT</td>
<td>opt</td>
</tr>
<tr>
<td>T_ORDER</td>
<td>order</td>
</tr>
<tr>
<td>T_OUT</td>
<td>out</td>
</tr>
<tr>
<td>T_PAR</td>
<td>par</td>
</tr>
<tr>
<td>T_PROCESS</td>
<td>process</td>
</tr>
<tr>
<td>T_REFERENCE</td>
<td>reference</td>
</tr>
<tr>
<td>TRELATED</td>
<td>related</td>
</tr>
<tr>
<td>T_RESET</td>
<td>reset</td>
</tr>
<tr>
<td>T_SEQ</td>
<td>seq</td>
</tr>
<tr>
<td>T_SERVICE</td>
<td>service</td>
</tr>
<tr>
<td>T_SET</td>
<td>set</td>
</tr>
<tr>
<td>T_SHARED</td>
<td>shared</td>
</tr>
<tr>
<td>T_STOP</td>
<td>stop</td>
</tr>
<tr>
<td>T_SUBST</td>
<td>subst</td>
</tr>
<tr>
<td>T_SYSTEM</td>
<td>system</td>
</tr>
<tr>
<td>T_TEXT</td>
<td>text</td>
</tr>
<tr>
<td>T_TIMEOUT</td>
<td>timeout</td>
</tr>
<tr>
<td>T_TO</td>
<td>to</td>
</tr>
<tr>
<td>T_VIA</td>
<td>via</td>
</tr>
</tbody>
</table>
Appendix C List of C Functions

MSC_RETURN_TYPE MSC lx_init_1(FILE *p_input_file)
  // initialize the read file buffer and global variables

static int MSC lx_read_bf(FILE *p_input_file, char *p_buffer)
  // read character string from file to the buffer

static MSC RETURN_TYPE MSC init_buffer(FILE * p_input_file)
  // Read the buffer at the initial time

char MSC lx_getchar(FILE *p_input_file)
  // gets next character from buffer. return null if it is end of file

char MSC lx lookahead()
  // Function looks next available char

void MSC lx_get_string(FILE *p_input_file, char *p_out_string)
  // gets the string up to end of line or end of file.

static void MSC init alphanumeric()
  // initialize alphanumeric char array used to check a char is alphanumeric or not

int MSC lx_S Alphanumeric(char p_ch)
  // check a char is alphanumeric or not

static void MSC lx_get alphanumeric_string(FILE *p_input_file, char *p_char, int * p_length)
  // get alphanumeric char string followed by p_char from the buffer

static MSC RETURN_TYPE MSC lx get word(FILE *p_input_file, char *p_string, int *p_length )
  // get the pointer and length of a word string start with p_char.

MSC RETURN_TYPE MSC lx_get_name(FILE *p_input_file, char *p_string)
  // gets a name string. Name string is defined by MSC

void test main()
  // test main function to run the tool

static int MSC lx_init(FILE *p_input_file)
  // Overall initialize function

FILE *MSC get file_ptr()
// Return current file pointer

char *MSC_get_name()
    // Return current name string of an identifier

static void MSC_lx_skip_wsp(FILE *p_input_file, char p_ch)
    // skip white space

MSC_token_td MSC_lx_get_token()
    // get next token

static MSC_token_td MSC_lx_letter(FILE *p_input_file)
    // lexical function used to analysis alphanumeric char start string
    // It is call MSC_lx_get_name to get a name

static int MSC_compare_key_word(const void *p_keyv_ptr, const void *p_table_entry_ptr)
    // compare a name string against a key enter in the word table. Like comp function

static MSC_token_td MSC_lx_check_key(char *p_string)
    // check a string is keyword or not.

static MSC_token_td MSC_lx_number(FILE *p_input_file)
    // lexical analyze for a string started with digit number. Not used in the project

static MSC_token_td MSC_lx_just_return(FILE *p_input_file)
    // Return token assigned to the character

    Please note: following parser function also build the represent classes object.

int MSC_ps_is_end_token(MSC_token_td p_token)
    // check a token is <end>. This design is intent to add comment before the end

void MSC_ps_end()
    // parser <end> rule 1

void MSC_ps_msc_document()
    // parser grammar rule 2-7

void DrawSystem()
    // start draw.

void SetViewPtr(CMscGraphicView* Vp )
    // update view pointer of the object tree.

void MSC_ps_message_sequence_chart()
void MSC_ps_msc_head()
    // parser grammar rule 9, 10

void MSC_ps_msc_inst_interface()
    // parser grammar rule 11

void MSC_ps_instance_list()
    // parser grammar rule 12

void MSC_ps_instance_kind()
    // parser grammar rule 13, 14

void MSC_ps_msc_gate_interface()
    // parser grammar rule 15

void MSC_ps_msc_gate_def()
    // parser grammar rule 16-18 and part of 49, 50 and rule 55

void MSC_ps_def_in_gate()
    // parser grammar rule 49

void MSC_ps_def_out_gate()
    // parser grammar rule 50

void MSC_ps_def_order_in_gate()
    // parser grammar rule 54

void MSC_ps_msg_identification()
    // parser grammar rule 40 and 41

void MSC_ps_input_dest()
    // parser grammar rule 47

void MSC_ps_input_address()
    // parser grammar rule 44

void MSC_ps_output_dest()
    // parser grammar rule 48

void MSC_ps_output_address()
    // parser grammar rule 42

void MSC_ps_input_output_address()
    // sub-function parser grammar rule 42, 44
void MSC_ps_order_dest()
  // parser grammar rule 52

void MSC_ps_reference_identification()
  // parser grammar rule 43

void MSC_ps_name_list(MSC_NAME_TYPE p_type, MSC_NAME_STATUS p_status)
  // sub function to parser a name list, like <name> [,<name>]*

int MSC_ps_is_expr_token(MSC_token_td p_token)
  // function to check if the token is a inline token

void MSC_ps_init_instance_no_type()
  // initialize function to the stack of structure, which is used to check single or multi
  // instances.

int MSC_ps_is_single_inst()
  // Return true if the events belong to a single(one) instance

int MSC_ps_is_multi_inst()
  // Return true if the events belong to multi (more than one) instances

void MSC_ps_reset_inst_no_type(MSC_INST_NO_TD p_inst_no_td)
  // reset the stack

void MSC_ps_increase_inst_type_index()
  // increase the stack pointer

void MSC_ps_decrease_inst_type_index()
  // decrease the stack pointer.

void MSC_ps_put_back_token(MSC_token_td p_token)
  // put the token into a temporary stack

void MSC_ps_get_store_token()
  // get the token in the temporary storage or get one from the file

int MSC_is_any_store_token()
  // return is there any stored token

int MSC_update_instances(MSC_NAME_STATUS p_status)
  // update the instance status.

void MSC_add_all_instance_list(char *p_name)
  // add an instance to all instance list

56
void MSC_ps_msc_body()
    // parser grammar rule 19

void MSC_ps_msc_statement()
    // parser grammar rule 20

void MSC_ps_text_definition()
    // parser grammar rule 21

void MSC_lx_get_text_string(char *p_text_string)
    // get text string

void MSC_ps_old_instance_head_statement()
    // parser grammar rule 31

void MSC_ps_event_definition()
    // parser grammar rule 22

void MSC_ps_decomposition(char *p_inst_name)
    // parser grammar rule 109 and 110

void MSC_ps_instance_event_list()
    // parser grammar rule 23

void MSC_ps_instance_event()
    // parser grammar rule 24

void MSC_ps_orderable_event(int p_inside_one_inst)
    // parser grammar rule 25

void MSC_ps_message_event()
    // parser grammar rule 34 and 37

void MSC_ps_message_output()
    // parser grammar rule 35 and 38

void MSC_ps_message_input()
    // parser grammar rule 36 and 39

void MSC_ps_create()
    // parser grammar rule 71

void MSC_ps_time_statement()
    // parser grammar rule 66 to 69
void MSC_ps_action()
    // parser grammar rule 71

void MSC_ps_multi_instance_event_list()
    // parser grammar rule 29

void MSC_ps_multi_instance_event()
    // parser grammar 30

void MSC_ps_non_orderable_event()
    // parser grammar 27

void MSC_ps_coregion()
    // parser grammar 73 NS 74

void MSC_ps_condition()
    // parser grammar 65

void MSC_ps_msc_reference()
    // parser grammar 92, 93 AND 94

void MSC_ps_msc_ref_expr()
    // parser grammar 95

void MSC_ps_msc_ref_par_expr()
    // parser grammar 96

void MSC_ps_msc_ref_seq_expr()
    // parser grammar 97

void MSC_ps_msc_ref_exc_expr()
    // parser grammar 98

void MSC_ps_msc_ref_opt_expr()
    // parser grammar 99

void MSC_ps_msc_ref_loop_expr()
    // parser grammar 100

void MSC_ps_parameter_substitution()
    // parser grammar 101

void MSC_ps_substitution_list()
    // parser grammar 102

void MSC_ps_substitution()
    // parser grammar 103, 104, 105, and 106
void MSC_ps_ref_gate()  
    // parser grammar 107 and 108

void MSC_ps_actual_out_gate()  
    // parser grammar 45

void MSC_ps_actual_in_gate()  
    // parser grammar 46

void MSC_ps_actual_order_out_gate()  
    // parser grammar 51

void MSC_ps_inline_expr()  
    // parser grammar 75 and 81

void MSC_ps_loop_expr()  
    // parser grammar 76 and 82

void MSC_ps_opt_expr()  
    // parser grammar 77 and 83

void MSC_ps_exc_expr()  
    // parser grammar 78 and 84

void MSC_ps_alt_expr()  
    // parser grammar 79 and 85

void MSC_ps_par_expr()  
    // parser grammar 80 and 86

void MSC_ps_inline_expr_term(MSC_token_td p_token, MSC_INLINE_TYPE p_type,  
char * p_function_name )  
    // sub function parser common part of each inline expression and grammar 89

void MSC_ps_inline_gate_interface()  
    // parser grammar 90

void MSC_ps_inline_gate_interface_term()  
    // parser grammar 91

void MSC_ps_inline_out_gate()  
    // parser grammar 56

void MSC_ps_inline_in_gate()  
    // parser grammar 57
void MSC_ps_inline_order_out_gate()
  // parser grammar 58

void MSC_ps_inline_order_in_gate()
  // parser grammar 59

void MSC_ps_instance_head_statement()
  // parser grammar 32

void MSC_ps_instance_end_statement()
  // parser grammar 33

void MSC_ps_stop()
  // parser grammar 72

void MSC_ps_shared(SHARED_TYPE p_shared_type)
  // parser grammar 63 and 64

void MSC_ps_loop_boundary(MSC_NAME_TYPE p_type)
  // parser grammar 87 and 88

void MSC_call_msc_body()
  // function pushes other variables before call msc_bosy

void MSC_ps_msc_expression(HMSC_expression *p_hmsc)
  // parser grammar 111

void MSC_ps_start(HMSC_expression *p_hmsc)
  // parser grammar 112

void MSC_ps_node_expression(HMSC_expression *p_hmsc)
  // parser grammar 113

void MSC_ps_node(HMSC_expression_node *p_expr_node)
  // parser grammar 114

void MSC_ps_par_expression(HMSC_node *p_node)
  // parser grammar 115

Following is the utility functions, which are used to build the objects

void MSC_print_token(MSC_token_td p_token)
  // print token name by token number

int MSC_ps_insert_message()
// Create and insert a message object into local table

message * MSC_ps_create_message()
    // create message object

void MSC_ps_insert_order_event(char *p_name, order_inst_event *p_order_event)
    // insert the order inst event into the local table

int check_inst_in_table(char *p_name, MSC_NAME_STATUS p_status)
    // check the instance status by using local table

void MSC_ps_update_inst_in_table(char *p_inst_name, instance *p_instance)
    // update the instance definition in the local table

void MSC_ps_update_instances_event(MSC_element *p_event)
    // Add event to each instance on the current instance list

Instance_List *MSC_ps_getInst_ptr_list(CStringList& p_name_list)
    // return an instance pointer list by a name list
Appendix D  Class Dictionary

1 Class MSC_element

typedef enum {LINE,RECTANGLE} CEElement_shape;

class MSC_element : public CObject
{
    DECLARE_DYNAMIC(MSC_element)
public:
    virtual void draw(void) const
    
    virtual void print_item(void) const
    
    virtual void update_DrawNo(int DrawNo)
    
    virtual void update_pos(MSC_element *PriorElementPtr,
        long Hunit,long Vunit)
    
    virtual void update_ViewPtr(CMscGraphicView* ptr)(m_ViewPtr = ptr);

    MSC_element();
    MSC_element(char *p_string);
    inline int operator == (const MSC_element &e) const {
        return (element_name == e.element_name);
    }

protected:
    CString element_name;
    CEElement_shape m_shape;

public:
    int m_DrwOdrNo;
    CMscGraphicView* m_ViewPtr;
    CPoint m_AxisStart;
    CPoint m_AxisEnd;
};

typedef CList<MSC_element*, MSC_element*> MSC_Element_List;

2 Class msc_document

class msc_document : public MSC_element
{
    DECLARE_DYNAMIC(msc_document)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
virtual void update_DrNo(int DrawNo);
virtual void update_pos(MSC_element *PriorElementPtr,long Hunit,long Vunit);
virtual void update_ViewPtr(CMscGraphicView* ptr);

MSC_element(char *p_name) { m_DrwOdrNo = 0; } ;
void add_basic_msc(msg_seq_chart *p_msc) {
    msc_seq_chart_list.AddTail(p_msc); }

void add_high_msc(HMSC_expression *p_hmsc) {
    HMSC_expression_list.AddTail(p_hmsc); }

private :
    Msg_Seq_Chart_List msc_seq_chart_list;
    HMSC_Expression_List HMSC_expression_list ;

};

3 Class msg_seq_chart

class msg_seq_chart : public MSC_element
{
    DECLARE_DYNAMIC(msg_seq_chart)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrNo(int DrawNo);
    virtual void update_pos(MSC_element *PriorElementPtr,long Hunit,long Vunit);
    virtual void update_ViewPtr(CMscGraphicView* ptr);

    msg_seq_chart() : MSC_element(), instance_list() { m_DrwOdrNo = 0; } ;

    msg_seq_chart(char * p_msg_seq_chart_name)
        : MSC_element(p_msg_seq_chart_name), instance_list()
        { m_DrwOdrNo = 0; } ;

    void add_instance(instance *p_instance);
    void add_gate(msc_gate *p_gate);

    friend class MSC_table ;
    friend class CMscGraphicView;

private:
    Instance_List instance_list ;
    Gate_List def_gate_list ;
};
typedef CList<msg_seq_chart*, msg_seq_chart*> Msg_Seq_Chart_List;

4 Class instance

class instance : public MSC_element
{
    DECLARE_DYNAMIC(instance)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_pos(MSC_element *PriorElementPtr,long Hunit,long Vunit);
    virtual void update_DrNo(int DrawNo);
    virtual void update_ViewPtr(CMscGraphicView* ptr);

    instance() : MSC_element(), kind_denominator(), kind_name(),
                inst_event_list(), decomposition_name()
    { decomposition_ptr = NULL ; m_DrwOdrNo =0;  } ;

    instance(char * p_inst_name) : MSC_element(p_inst_name), kind_denominator(),
                                  kind_name(), inst_event_list(), decomposition_name()
    { decomposition_ptr = NULL ; m_DrwOdrNo = 0; } ;

    int update_instance(const instance &p_inst);

    void update_kind_denominator(char *p_kind_denominator);
    void update_dmom_name(char *p_dmom_name);
    void update_decom_name(char *p_name) { decomposition_name = p_name ; } ;

    void add_inst_event(MSC_element *p_event);
    void print_name() { printf("inst name is %s \n", element_name); ; }

    friend MSC_table ;
    friend message ;

private:
    CString kind_denominator ;
    CString kind_name ;
    MSC_Element_List inst_event_list ; // point to all event belonged to this instance
    msg_seq_chart *decomposition_ptr ;
    CString decomposition_name;

    CPoint m_InstStart1;
    CPoint m_InstStart2;

    CPoint m_InstEnd1;
    CPoint m_InstEnd2

64
typedef CList<instance*, instance*> Instance_List;

5 Class order_inst_event

typedef order_inst_event* Order.Inst_Event_Ptr;
typedef CList<Order.Inst_Event_Ptr, Order.Inst_Event_Ptr> Order.Inst_Event_List;

class order_inst_event : public MSC_element
{
    DECLARE_DYNAMIC(order_inst_event)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrNo(int DrawNo);
    virtual void update_pos(MSC_element *PriorElementPtr, long Hunit, long Vunit);
    virtual void update_ViewPtr(CMscGraphicView* ptr);

    order_inst_event();
    order_inst_event(char *p_name, MSC_element *p_element);

    void add_after_event_name(CStringList *p_event_name_list);

    friend MSC_table;

private:
    MSC_element *event;
    CStringList after_event_name;
    Order.Inst_Event_List after_event;
};

6 Class Timer

typedef enum
{
    UNKNOWN_TIME,
    SET_TIME,
    RESET,
    TIME_OUT
} ACT_TIMER_TYPE;

class msc_timer : public MSC_element
{
    DECLARE_DYNAMIC(msc_timer)
public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrawNo(int DrawNo);
    virtual void update_pos(MSC_element *PriorElementPtr, long Hunit, long Vunit);
    virtual void update_ViewPtr(CMscGraphicView* ptr)(m_ViewPtr = ptr);

private:
    ACT_TIMER_TYPE act_type;
    CString duration;
    CPoint m_TimerStartPoint;
    CPoint m_TimerEndPoint;
};

7 Class Create Instance

class create_instance_class : public MSC_element
{
    DECLARE_DYNAMIC(create_instance_class)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrawNo(int DrawNo) {};
    virtual void update_pos(MSC_element *PriorElementPtr,
                            long Hunit, long Vunit) {};

    create_instance_class();
    create_instance_class(char *p_name);
    create_instance_class(char *p_name, CStringList *p_para_list);
    void update_inst_ptr(instance *p_inst);

private:
    CString created_instance_name;
    instance *created_instance;
    CStringList inst_parameter_list;
};

8 Class msc_action
class msc_action : public MSC_element
{
    DECLARE_DYNAMIC(msc_action)

typedef enum {
    UNKNOWN_DIRECTION = 0,
    INST_TO_INST,
    ENV_TO_INST,
    INST_TO_ENV,
    REF_TO_INST,
    INST_TO_REF,
    INLINE_TO_INST,
    INST_TO_INLINE,
    TO_GATE,
    FROM_GATE
} MESSAGE_TYPE;

typedef enum {
    NOT_LOST,
    INPUT_LOST,
    OUTPUT_LOST
} MESSAGE_LOST_TYPE;

class message : public MSC_element
{
    DECLARE_DYNAMIC(message)

typedef enum {
    virtual void draw(void) const;
    virtual void print_item(void) const;

    public:
    virtual void update_DrNo(int DrawNo) {}
    virtual void update_pos(MSC_element *PriorElementPtr,
        long Hunit, long Vunit) {};

    msc_action(char *p_name) : MSC_element(), action_string(p_name) {};

private:
    CString action_string;
};

9 Class message

9 Class message
virtual void update_DrawNo(int DrawNo);  
virtual void update_pos(MSC_element *PriorElementPtr, long Hunit, long Vunit);  
virtual void update_ViewPtr(CMscGraphicView* ptr){m_ViewPtr = ptr;};

message();
message(char *p_string, MESSAGE_TYPE p_type,
    MESSAGE_LOST_TYPE p_lost_type, char *p_from_name,
    char *p_to_name, char *p_gate);

void add_from_element(MSC_element *p_from_inst)
    { from_inst = p_from_inst; };
void add_to_element(MSC_element *p_to_inst) { to_inst = p_to_inst; };

friend int operator == (const message &msg1, const message &msg2 );
void add_parameter(CStringList *p_list);

friend MSC_table  ;

public:
    MESSAGE_TYPE message_type  ;

private:
    MSC_element *from_inst  ;
    MSC_element *to_inst  ;
    CString from_name;
    CString to_name;
    CStringList parameter_list;
    MESSAGE_LOST_TYPE lost_type;

    CString via_gate_name;
    msc_gate *via_gate;

    CPoint m_MsgStart;
    CPoint m_MsgEnd;
};

10 Class simple_event

typedef enum {
    COREGION_START,
    COREGION_END,
    INSTANCE_HEAD,
    INSTANCE_END,
    INSTANCE_STOP,
};

68
class simple_event : public MSC_element
{
    DECLARE_DYNAMIC(simple_event)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrawNo(int DrawNo);
    virtual void update_pos(MSC_element *PriorElementPtr,long Hunit,long Vunit);
    virtual void update_ViewPtr(CMscGraphicView* ptr){m_ViewPtr = ptr;};

    simple_event(MSC_SIMPLE_EVENT_TYPE p_type):
        MSC_element() { event_type = p_type; };

private:
    MSC_SIMPLE_EVENT_TYPE event_type;
    CPoint m_SmpevtStart;
    CPoint m_SmpevtEnd;
};

11 Class condition

class MSC_table ;

class condition : public MSC_element
{
    DECLARE_DYNAMIC(condition)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrawNo(int DrawNo);
    virtual void update_pos(MSC_element *PriorElementPtr,long Hunit,long Vunit);
    virtual void update_ViewPtr(CMscGraphicView* ptr);

    condition() : MSC_element(), condition_name_list(), shared_instance_list()
    { m_DrawOrderNo = 0; m_OnceAccess = 0; };

    condition(CStringList& p_condition_name_list);

    void add_shared_instance(Instance_List* p_shared_instance);
friend MSC_table;

private:
    CStringList condition_name_list;
    Instance_List shared_instance_list;

    CPoint m_CdnStart;
    CPoint m_CdnEnd;
    int m.OnceAccess;

};

12 Class msc_inline

typedef enum {
    UNKNOWN_EXPR,
    LOOP_EXPR,
    OPT_EXPR,
    EXC_EXPR,
    ALT_EXPR,
    PAR_EXPR
} MSC_INLINE_TYPE;

class msc_inline : public MSC_element
{
    DECLARE_DYNAMIC(msc_inline)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrNo(int DrawNo);
    virtual void update_pos(MSC_element *PriorElementPtr,long Hunit,long Vunit);
    virtual void update_ViewPtr(CMscGraphicView* ptr){m_ViewPtr = ptr;};

    msc_inline();

    msc_inline(char *p_string, MSC_INLINE_TYPE p_type);
    void add_instancetxt(Instance_List *p_inst_list);
    void add_inline_gate1(msc_gate *p_gate);
    void add_inline_gate2(msc_gate *p_gate);

protected:
    MSC_INLINE_TYPE msc_inline_type;

private:
    Gate_List msc_gate_list1;
    Gate_List msc_gate_list2;
    Instance_List instance_list;
13 Class msc_loop_expr

class msc_loop_expr : public msc_inline
{
    DECLARE_DYNAMIC(msc_loop_expr)

    public:
    msc_loop_expr() : msc_inline(), lower_bound(), upper_bound()
    {
        msc_inline_type = LOOP_EXPR;
    };
    msc_loop_expr(char *p_string) :
    msc_inline(p_string, LOOP_EXPR), lower_bound(), upper_bound()
    {
        msc_inline_type = LOOP_EXPR;
    };

    void add_low_bound(char *p_bound);
    void add_up_bound(char *p_bound);
    void update_loop_name(char *p_string) {
        element_name = p_string;
    };

    virtual void print_item(void) const;

    private:
    CStringList lower_bound;
    CStringList upper_bound;
};

14 Class msc_reference

class msc_reference : public MSC_element
{
    DECLARE_DYNAMIC(msc_reference)

    public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrawNo(int DrawNo) {};
    virtual void update_pos(MSC_element *PriorElementPtr, long Hunit,
                            long Vunit) {};

    msc_reference(char *p_name, char *p_string);

    private:
    CString referr_item;
};

71
Class msc_gate

typedef enum {
    UNKNOWN_GATE,
    DEF_OUT_GATE,
    DEF_IN_GATE,
    ACTUAL_IN_GATE,
    ACTUAL_OUT_GATE,
    INLINE_IN_GATE,
    INLINE_OUT_GATE
} MSC_GATE_TYPE;

class msc_gate : public MSC_element
{
    DECLARE_DYNAMIC(msc_gate)

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrawNo(int DrawNo) {};
    virtual void update_pos(MSC_element *PriorElementPtr, long Hunit,
                            long Vunit) {};
    msc_gate();
    msc_gate(char *p_name, MSC_GATE_TYPE p_type);
    void update_related_msg(message *p_message);
    void updateExtern_msg(message *p_message);

    friend message;

public:
    MSC_GATE_TYPE msc_gate_type;

private:
    message *related_msg;
    message *extern_msg;
};

typedef CList<msc_gate*, msc_gate*> Gate_List;

Class HMSC_node

typedef enum {
    REF_NODE,
    EMPTY_MSC,
    MSC_NODE,
    PAR_HMSC,
    CONDITION,

    72
CONNECT,
END_NODE
} HMSC_NODE_TYPE;

class HMSC_expression;
typedef CList<HMSC_expression*, HMSC_expression*> HMSC_Expression_List;

class HMSC_node : public MSC_element
{
    DECLARE_DYNAMIC(HMSC_node);

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;

    HMSC_node(HMSC_NODE_TYPE p_type);
    HMSC_node(HMSC_NODE_TYPE p_type, char *p_string);
    HMSC_node(HMSC_NODE_TYPE p_type, CStringList *p_list);
    void add_par_hmsc(HMSC_expression *p_hmsc) {
        par_hmsc_list.AddTail(p_hmsc);
    }

private:
    HMSC_NODE_TYPE node_type;
    CStringList name_list;
    HMSC_Expression_List par_hmsc_list;

    int m_DrwOdrNo;
};

17 Class HMSC_expression_node

class HMSC_expression_node : public MSC_element
{
    DECLARE_DYNAMIC(HMSC_expression_node);

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;

    HMSC_expression_node(char *p_name);
    void add_after_label(char *p_label) { after_label.AddTail(p_label); }
    void assign_node(HMSC_node *p_node) { hmsc_node = p_node; }
    void add_after_list(HMSC_expression_node* p_expression_node);
private:
    HMSC_node *hmsc_node;
    CStringList after_label;
    HMSC_Expression_Node_List *after_list
    int m_DrwOdrNo;
};

typedef CList<HMSC_expression_node*, HMSC_expression_node*> HMSC_Expression_Node_List;

18 Class HMSC_expression

class HMSC_expression : public MSC_element
{
    DECLARE_DYNAMIC(HMSC_expression);

public:
    virtual void draw(void) const;
    virtual void print_item(void) const;
    virtual void update_DrNo(int DrawNo);{}
    virtual void update_pos(MSC_element *PriorElementPtr, long Hunit, long Vunit){};

    HMSC_expression() : MSC_element(), expression_label_list(), node_list() {};
    HMSC_expression(char *p_name)
        : MSC_element(p_name), expression_label_list(), node_list() {};

    void add_expression_label_list(char *);
    void add_node(HMSC_expression_node *);

private:
    CStringList expression_label_list;
    HMSC_Expression_Node_List node_list;
};

Following Table is used to build symbol table.

19 Class MSC_table

typedef enum {
    UNKNOWN_NAME,
MSC_NAME,
INST_NAME,
GATE_NAME,
MSG_NAME,
EVENT_NAME,
PARA_NAME,
TIME_NAME,
COND_NAME,
REF_NAME,
INLINE_NAME
} MSC_NAME_TYPE;

typedef enum {
    UNKNOWN = 0,
    REFERRED,
    USED,
    DEFINED,
    DECLARED,
    FINISHED
} MSC_NAME_STATUS;

typedef struct {
    char name[L_IDENTIFIER+1];
    MSC_NAME_TYPE type;
    MSC_NAME_STATUS status;
    MSC_element *object_ptr;
} MSC_Element_Node;

class MSC_table {
public:
    MSC_table();
    ~MSC_table();
    int insert_table(char *p_name, MSC_NAME_TYPE p_type,
                    MSC_NAME_STATUS p_status, int p_ignore_duplicate);
    int search_table(char *p_name);
    int update_table(char *p_name, MSC_element *p_ptr);
    int update_table(char *p_name, MSC_NAME_STATUS p_status);
    MSC_element *get_ptr(char *p_name);
    MSC_NAME_STATUS get_status(char *p_name);
    MSC_NAME_STATUS MSC_table::get_status(int p_position);
    MSC_NAME_TYPE get_type(int position);

    void update_event_link();
    void update_position();
    void print_table();

75
private:
    int hashkey1(char *p_str, int M);
    int hashkey2(char *p_str);
private:
    MSC_Element_Note element_table[TABLE_SIZE];
    CElement* m_pTempElement;