Chapter 8

Pragmatics: Prototyping Multimedia Applications

MICE is a multimedia development environment for the rapid prototyping of multimedia applications. The traditional "waterfall" software life cycle model is depicted in Figure 1.

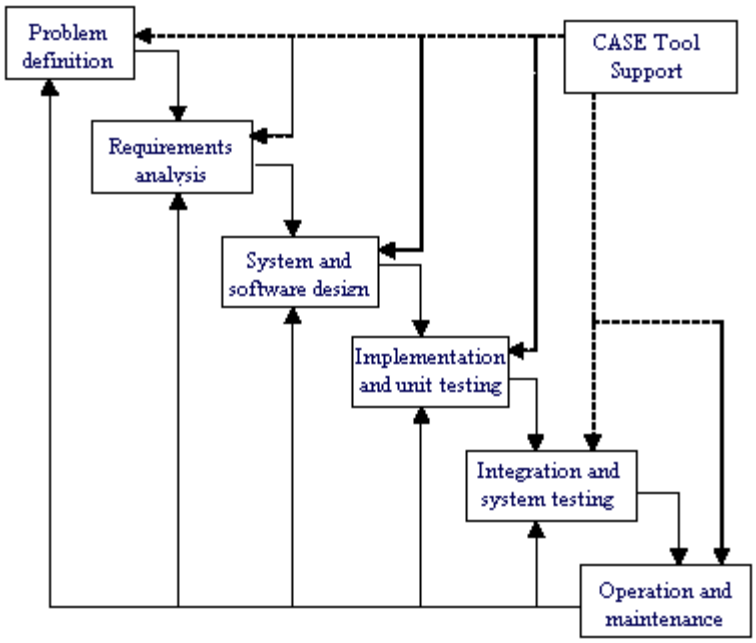

Figure 1. Traditional "waterfall" software life cycle model

This traditional software life cycle is appropriate for traditional application software development. Multimedia applications, on the other

hand, place strong emphasis on evolutionary content development. The rapid prototyping model is depicted in Figure 2.

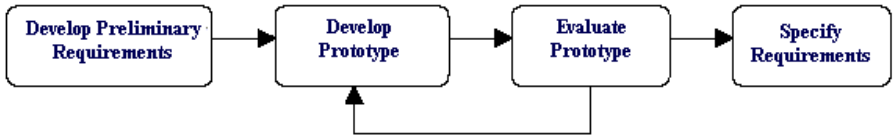

Figure 2. Rapid prototyping model for software development.

MICE is an application software development environment supporting rapid prototyping. In this chapter the details of MICE will be explained.

1. HOW TO BUILD A MICE APPLICATION

MICE provides a logical way in building a multimedia application on a workstation or a PC. From the web site www.cs.pitt.edu/~chang, by following the links to multimedia software engineering courseware, you will be led to the following directories that contain the essential components of MICE:

- IC_Builder/ the files you need to run the IC_Builder on PC
- IC_Compiler/ the files to run the IC_Compiler
- IC_Manager/ the files needed to compile the IC_Manager
- IC_Taoml/ the interpretor to translate .taoml pages to .html pages

The seven steps to build a MICE application in a project directory such as IC_Work/ are described below:

Step 1. Download the IC_Builder to your PC, unzip and install it under Windows. Use IC_Builder (see Section 2) to draw each index cell and create the .in file for each ic. Use capital letters for the *.in files such as XIC.in, DIC.in, etc. The IC_Builder will also create the ic.dat file. You can also create *.in and ic.dat manually without using IC_Builder.

Step 2. Provide one action file for each action defined in the ic's. An action file contains the corresponding C function for each action and should be copied to the IC_Work/source/ directory.

Step 3. Copy *.in files to IC_Work/ directory and ic.dat file to IC_Work/source/ directory. Copy all files from IC_Compiler, IC_Manager and IC_Taoml to IC_Work/source/ directory. If necessary, modify the ic.dat file. Use IC_Compiler icc to generate the source files. There are six files generated by IC_Compiler:

actions.c ic_func2.c ic_func3.c app.h fuzzy.h db_def.h

Step 4. Use command "make -f makefile.maincgi" to make main.cgi that is the cgi program that your application will need. Therefore, you may call it main.cgi. The IC_Manager becomes part of main.cgi so any message to an ic will be sent to this main.cgi.

Step 5. Use command "make -f makefile.intercgi" to make inter.cgi that is the cgi program to access a taoml page. main.cgi and inter.cgi should be copied to application directory IC_Work/ so that the home page can use inter.cgi to access a taoml page such as tao_1.taoml. The link has the following form: <a href="inter.cgi?tao_name=tao_1.taoml">.

Step 6. Design the home page index.html for the application and put it in directory IC_Work/. This home page should have a cgi link to a taoml page such as tao_1.taoml. tao_1.taoml and its associated template page tao_1.tpl should be in the sub-directory IC_Work/TAOML/. Indeed, all taoml pages and tpl pages should be in the sub-directory IC_Work/TAOML/. To create the taoml pages, you will use an extended html syntax to specify the TAOs and how they are structured and activate ic using the cgi program, which is main.cgi. If you want to refer to your own cgi programs, they can be mentioned in the tpl pages. To sum up, there are three types of pages:

- html page index.html uses cgi program inter.cgi to link to tao_1.taoml
- taoml page tao_1.taoml uses cgi program main.cgi to activate an ic tao_1
- tpl page tao_1.tpl uses customized cgi program to do special tasks

Step 7. Now you are ready to run your application. Use a browser to enter application's home page IC_Work/index.html.

Notes:

- For detailed step-by-step instructions, see Section 6, MICE Application Development Steps.
- Program compilation must be done on the same type of computer system as the server.
- In IC_Compiler directory, the action template is action.tpl and the customized action functions are a1.c, a2.c, ..., etc. which correspond to the actions a1, a2, ..., etc.

2. IC BUILDER

The IC Builder is a PC-based tool to help the user define active index cells. Once an ic is defined, the IC Builder creates a formal specification file *.in such as ic1.in. After all the ic's have been defined, the IC Builder generates a file ic.dat to characterize an application. This file ic.dat becomes the input to the next tool, the IC Compiler.

The ic specification files *.in, on the other hand, become the input to the customized IC Manager.

The Symbolic IC_Builder Version 2.0 has the following features:

- There is no need to specify the message ID and action ID any more, i.e. messages and actions are directly represented by symbolic names.
- The ic.dat file will be automatically generated.
- Simple project management for all the ICs in the project.

The following steps will create the ICs and .in and ic.dat files:

Step 1. Create a project directory, which will contain all the project files later, such as c:\icb\hw4\

Step 2. In the IC Builder menu bar, find the 'simulation' menu. Select the 'options' menu item. This will bring up a dialog asking you to specify the executable application file (the customized IC Manager) and the message input file. For example, if your customized IC Manager is an executable file called wag.exe, then you first enter:

c:\icb\hw4\wag.exe

and then you enter:

c:\icb\hw4\mag.in

which means the message input file is msg.in. These two inputs are served as the simulation purpose in IC_Builder. But they are also used to determine the project directory, which will be subtracted from the executable file path. This means all the files generated by IC_Builder will be put in this directory.

Step 3. Also in the 'simulation' menu, select the 'project' item to define the project files in the project. You could add or remove the files from the project. Please use the name of IC file, don't use the .in name. For example, if your project contains three ics: WAG, BBC, LBC, then the project files could be:

WAG.gra BBC.gra, LBC.gra

Step 4. Use the IC_Builder to create ICs. The input message and output message specification dialogs are explained later in this section.

Step 5. After you have created all the ICs, click the 'export' button to export the .in files and ic.dat. IC_Builder will create .in file for each IC, the name of .in file is same as its graphic file.

The key features of the IC_Builder are described below:

2.1. Define a Project

2.1.1 Create a Project Directory as Your Work Space: Create a directory in the disk, which will contain all the project files later. For example, c:\icb\hw4\.

2.1.2 Specify Project Files: In the IC Builder menu bar, find the 'simulation' menu. Select the 'options' menu item. This will bring up a dialog which ask you to specify the executable file for the customized IC Manager of the application and the external input message file if necessary. Please be

sure to specify the complete path of the project directory so that any file generated by IC_Builder will be put in this directory.

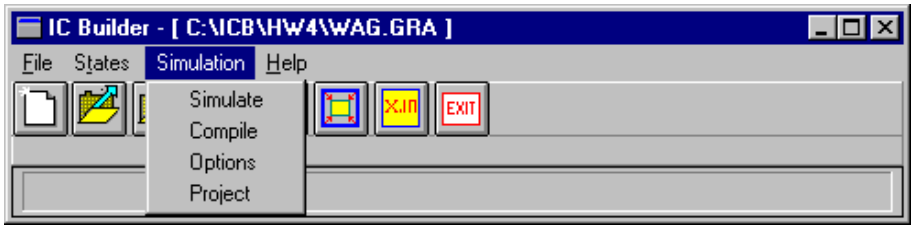

Figure 3. How to specify project files.

For example, if your customized IC Manager is an executable file called wag.exe, then at the first line input the following:

c:\icb\hw4\wag.exe

and the second line can be something like:

c:\icb\hw4\mag.in

which means the external input message file is msg.in.

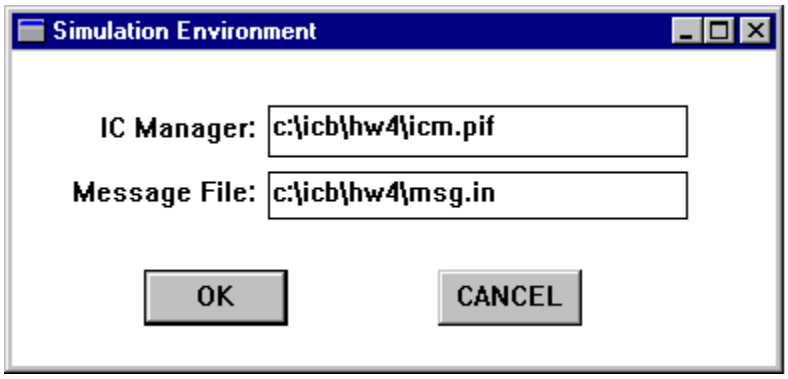

Figure 4. How to specify IC Manager and message file.

2.1.3 Specify IC Types used in the Project: Also in the 'simulation' menu, select the 'project' item to define the project files in the project. You could add or remove the files from the project. Please use the name of IC file, don't use the .in name. For example, if your project contains three ics: WAG, BBC, LBC then the project files could be: WAG.gra BBC.gra, LBC.gra.

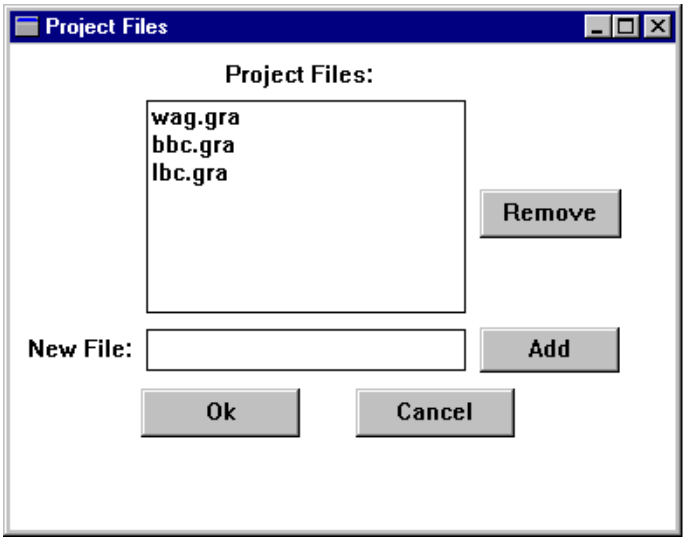

Figure 5. The project files.

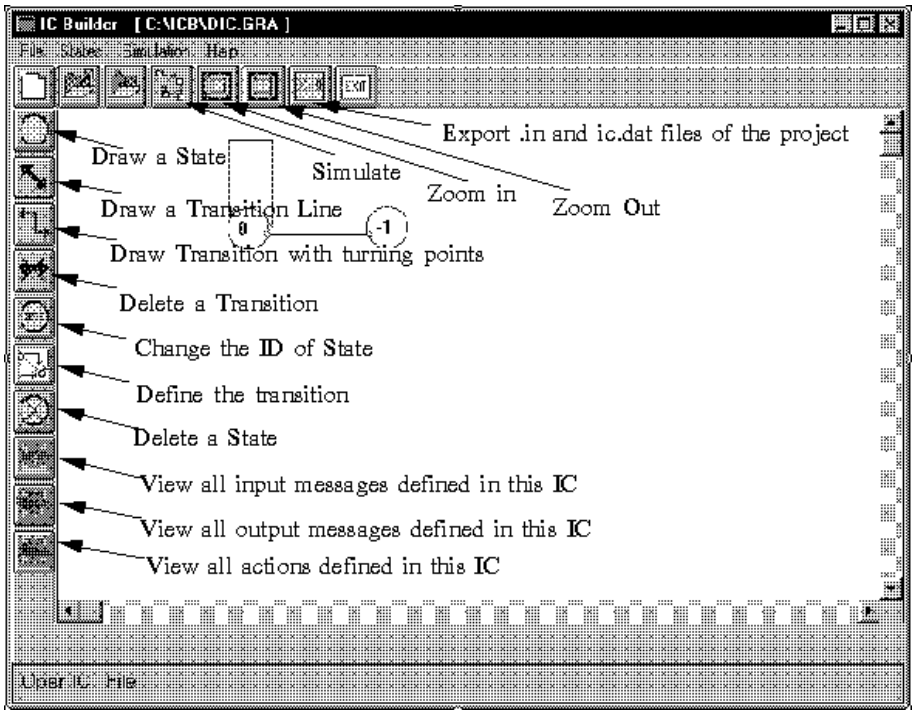

Figure 6. The IC Builder's tool bar.

2.2. Define IC Types

2.2.1 Draw a State: Click (press left mouse button) the icon in the tool bar, then point the cursor at the desired position, press left button. The state will be numbered automatically.

2.2.2 Delete a State: Click (press left mouse button) the Delete_State icon in the tool bar, then move the cursor within the state which you want to delete, press left button.

2.2.3 Move a State: Currently, there is no way to move a state to a different position, you have to delete and draw a new one to achieve the reposition of the state.

2.2.4 Change the ID number of the State: Click (press left mouse button) the Change_ID icon in the tool bar, then click on the state you want to change. A dialog will appear to let you enter the new number of the state.

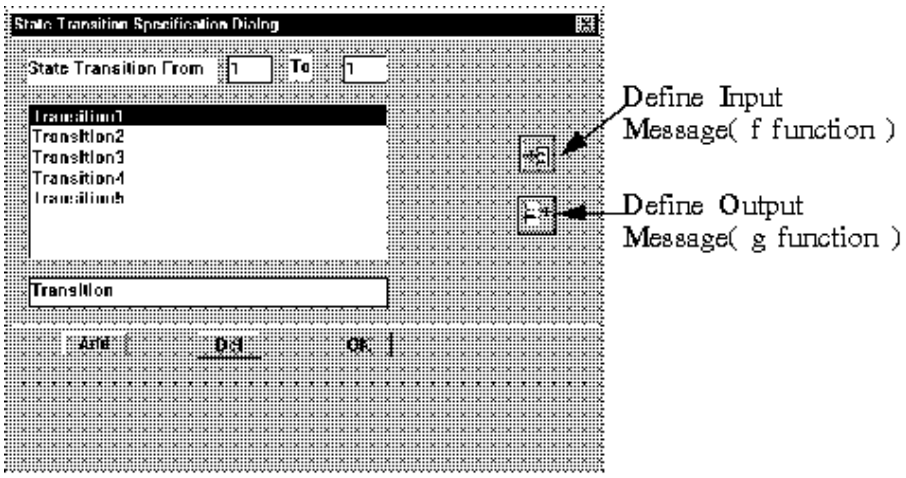

Figure 7. How to define a transition.

2.2.5 Draw a Transition: There are two ways to draw a transition between states. One way is to draw a straight line, the other is to draw a line with turning points. Either way you should first click the icon, then move the cursor to the transition start position on one state, then click. Then you can move the cursor to the next position, click, and so on (if you are not using the draw straight line icon). Finally you double click the left button to select the end position of the transition. Be aware that the start and end positions should always be on the edge of the state. The start position is marked as a small green rectangle, and the end position is marked as a red rectangle.

2.2.6 Delete a Transition: Select the Delete_Transition icon in the tool bar, move the cursor to the start position of the transition, then click the left button.

2.2.7 Define a Transition: Click the Define_Transition icon in the tool bar, then move the cursor to the start position of one transition, click the left button. A dialog like above will appear on the screen. This dialog lets you to add as many transitions between two same states as you want. Click the two buttons on the right of the dialog to further define the input or output message of one transition between the two states.

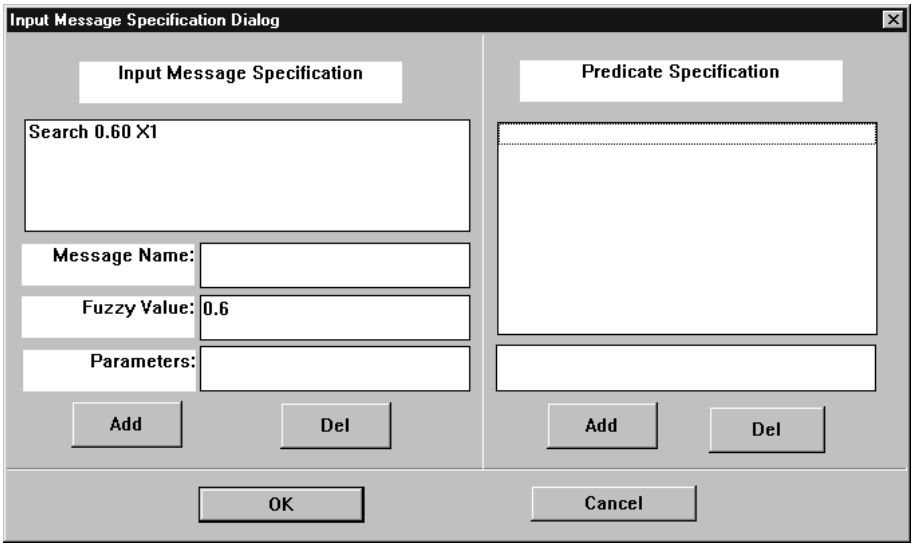

Figure 8. How to define the input message.

2.2.7.1 Input Message Specification Dialog: As you click the Define Input Message button, the Input Message Specification dialog will prompt, as the figure below. There are two columns in the dialog, the left one is used to specify input message's name, parameters. The right column is used to define the predicate for input messages. You can add or delete the input message and predicate. The format of the input field "Parameters" will be explained later in Section 2.3. Notice the message name is case insensitive.

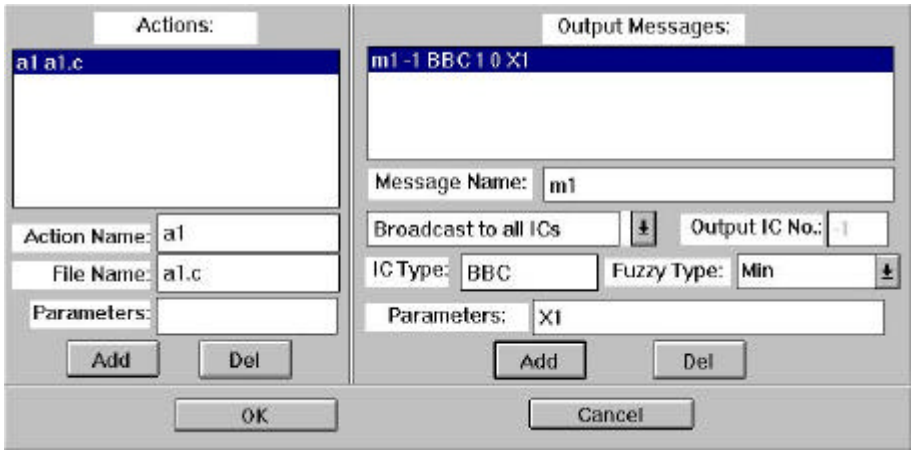

Figure 9. Output message specification.

2.2.7.2 Output Message Specification: There are two columns in this dialog, the left one is used to define the action for the transition. Two fields are needed for each action, the action name (case insensitive) and the name of the file that contains the action. The right column defines the output messages in the transition. The format of the input field "Parameters" in the action and output message will be explained later in Section 2.3. There are six options for the field "Output IC NO.":

- Specify an Existing IC ID: For this option, the user has to specify a positive integer as the IC ID.
- Send to a New IC: For this option, the corresponding output message will be post to an IC which will be activated when this message comes. Please notice you must specify the IC type in the field "IC type".
- Broadcast to All ICs: For this option, this message will be broadcast to all ICs. If the IC type in the field "IC type" is specified, the message will be broadcast to all ICs of the specified type. If not, the message will broadcast to all ICs that can receive the message.
- Contended by All ICs: For this option, this message will be contended by all ICs. If the IC type in the field "IC type" is specified, the message will be contended by all ICs of the specified type. If not, the message will be contended by all ICs which can receive the message.
- Broadcast to Selected ICs: For this option, this message will be broadcast to the selected ICs. The user has to program a function to compute the selected ICs. If the function needs to know the IC type, the user has to the IC type in the field "IC type".
- Contended by Selected ICs: For this option, this message will be contended by the selected ICs. The user has to program a function to

compute the selected ICs. If the function needs to know the IC type, the user has to the IC type in the field "IC type".

2.2.8 Export the IC diagram: The IC diagrams definition need to be transformed to so called .in file when it is used as the input of the IC_Manager. Click the Export icon in the tool bar to export the diagram as a .in file. The export operation will create all the .in files in the project plus an ic.dat file for the input of IC compiler.

2.3. The format of the parameter in the Message Definition Dialog is given below in BNF syntax:

<para_list> ::= <para_list>'|'<item>
<para_list> ::= <item>
<item> ::= <const> | <var> | <func>
<const> ::= I<integer> | F<float> | S<string>
<var> ::= X<digit> | Y<digit> | Z<digit>
<func> ::= G<digit>'('<func_para_list>')' | H<digit>'('<func_para_list>')'
<func_para_list> ::= <f_para_list> | NULL
<f_para_list> ::= <f_para_list>','<item>
<f_para_list> ::= <item>
<digit> ::= '0'..'9'

For example, X1|Y1|G7(X2, Y2) in the field parameters means that X1 and Y1 are variable parameters and G7 is a function parameter which has two variable parameters X2 and Y2. I25|F1.2|Sfire means that there are three constant parameters: an integer 25, a floating point 1.2 and a string "fire".

2.4. A sample .in file is given below:

```
0       // current state
0       // next state
1       // 1 input message(s)
10:0,Y1|Y0 // message start_prefetch with 2 parameters
0       // no. predicate
0       // 0 output ic(s)
0       // 0 output message(s)
3       // 3 action(s)
11       // action "issue_proc"
12       // action "set_pid"
14,Y1|Shelp|H0(Y0) // action "compute_schedule" with 3 parameters
0       // current state
0       // next state
1       // 1 input message(s)
```

```
11:0      // message "end_prefetch"
0         // no. predicate
0         // 0 output ic(s)
0         // 0 output message(s)
1         // 1 action(s)
15        // action "set_pid_null"
0         // current state
-1        // next state
1         // 1 input message(s)
12:0      // message "kill_prefetch"
0         // no. predicate
0         // 0 output ic(s)
0         // 0 output message(s)
1         // 1 action(s)
13        // action "kill_proc"
```

3. IC COMPILER

The IC Compiler accepts an input file that characterizes an application and generates the customized source code of the IC Manager. The default input file is ic.dat produced by the IC Builder. The IC Compiler icc can be recompiled using the make file "makefile.icc".

Usage: icc [-d] input_file

The flag -d generate source codes with embedded debugging messages

Input of icc: The input_file specifies the characteristics of the application. The default input_file is ic.dat.

Output of icc: app.h, fuzzy.h, db_def.h, actions.c, ic_func2.c, and ic_func3.c.

Format of the input_file: Each definition type header must be prefixed by "$". All definition lines follow their definition type header without prefixed by any special character. A definition type must end with "%". A comment line must begin with "//". A space line is allowed.

The IC Compiler supports the following definition types:

(1) Header "$MESSAGE" defines input and output messages of IC with definition format:

message_name/message_id

This definition type is to generate message definitions in "app.h", message array msg[] in "fuzzy.h", and function decode_msg() in "ic_func3.c".

(2) Header "$INCLUDE_FILE" allows the user to adds including files to app.h with definition format:

file_name

(3) Header "$ACTION/AUTO_GEN:YES|NO" defines actions of IC with definition format:

action_name/action_id[/function_name[/file_name]]

This definition type is to generate action definitions in "app.h", function do_actions() and all action functions in "actions.c". If "actions.c" exists, the old "actions.c" will be moved to a backup file "actions.b*", for example, actions.b0, actions.b1....

If AUTO_GEN equals YES, actions.c will be automatically generated by collecting specified file_names; otherwise, all the file_names will be ignored and the user has to provide an actions.c by himself.

If AUTO_GEN equals YES but the file_name is not specified, a default template of the corresponding function will be inserted into the actions.c instead.

(4) Header "$IC_ID" defines IC_IDs with definition format:

name_of_ic_id/number

This definition type is to generate ic_id definitions in "app.h" and function decode_ic() in "ic_func3.c".

Note: ic_id EXTERNAL has been defined as -1 in "ic.h".

(5) Header "$MUST_FUNC/AUTO_GEN:YES|NO" defines functions that are necessary in IC Manager with definition format:

func_group_name[/file_name]

This definition type is to generate all functions of file "ic_func2.c". If "ic_func2.c" exists, the old "ic_func2.c" will be moved to a backup file "ic_func2.b*", for example, ic_func2.b0, ic_func2.b1....

If AUTO_GEN equals YES, ic_func2.c will be automatically generated by collecting all functions in specified file_names; otherwise, all the file_names will be ignored and the user has to provide an ic_func2.c by himself. If AUTO_GEN equals YES but file_name is not specified, a default template of the corresponding functions will be inserted into ic_func2.c in stead.

(6) *.tpl are the templates for $MUST_FUNC. The templates: out_msg.tpl, predicat.tpl, inter_mm.tpl, and func_var.tpl are default templates for functional groups FILL_OUTPUT_MSG_GROUP, PREDICATE_GROUP, INTERNAL_MM_GROUP, and USER_DEFINE_FUNC_VAR_GROUP, respectively. The customized

functions should be called out_msg.c, predicat.c, inter_mm.c, and func_var.c.

A default template for each group contains:

```
FILL_OUTPUT_MSG_GROUP: fill_content(), fill_itype()
PREDICATE_GROUP: pred_match()
INTERNAL_MM_GROUP: dump_internal_mm(), init_mm(),
        save_mm(), restore_mm()
USER_DEFINE_FUNC_VAR_GROUP: userdef_f(), userdef_v()
FILL_OUTPUT_IC_GROUP: find_ic()
```

It is recommended to copy and modify the default template functions for each function group.

(7) action.tpl is the template for user-supplied action functions, one for each action. The customized actions are stored in separate files such as a1.c, ..., a5.c.

(8) Header "$THRESHOLD" defines thresholds for fuzzy computation with definition format:

```
ic_type/fuzzy_number
```

Two arrays ic_type and threshold will be generated in "fuzzy.h".

Note: Definition type "$THRESHOLD" is necessary for fuzzy IC.

(9) Header "DB_ACCESS" defined the access of database with definition format:

```
view_name/database/tables/attributes/condition
```

where tables = table [,table]*
attributes = attribute [,attribute]*
condition is a predicate.

If database is "DEFAULT", it means to access the default database that is defined in the program. If attributes is "*", it is all attributes of the specified tables. If condition is "NULL", it means that the generated SQL has no condition. The definition will create a SQL command for the specified database:

```
SELECT attributes
FROM tables
WHEN condition
```

File db_def.h will be generated.

(10) Use makefile.icc to compile the IC Compiler. Usage:

```
make -f makefile.icc
```

Note: All definition types must always be specified, except $IC_ID.

4. THE IC MANAGER

This directory contains programs to use fuzzy IC manager.

Make File: The following make files contains paths that need to be changed, if necessary.

Makefile.maincgi: generate main.cgi that will trigger ICs.
Makefile.showcgi: generate show.cgi that will display all ICs.
Makefile.clearcgi: generate clear.cgi that will clear all ICs.

Header Files

I. Core Part: modules in this part should not be modified

ic.h: header file (constants and data structures) of the ic manager

II. Application Dependent Part: constants and data structures in the header files in this part should be customized according to the application.

app.h: constants and data structures of your application
fuzzy.h: the header file for message codes and thresholds of ic types
It is included in fuzzy.c.
mm.h: structures for internal memory of ic's
If you define mm.h, define the including of mm.h in ic.dat so that app.h will include mm.h.

C Files

I. Core Part: modules in this part should not be modified

main.c: the web-based driver functions.
ic_manager.c: ic manager
ic_functions.c: functions called by ic manager
util.c: functions to create C structures for f, g functions and messages to dump the content of various structures.
fuzzy.c: functions to implement fuzzy computation.
clear.c: a function to clear all ICs.
show.c: a function to display all current ICs.

II. Application Dependent Part: modules in this part should be customized according to the application. Examples and/or templates are provided for functions in each module.

driver.c: the text-based driver program
ic_func2.c: application dependent, but necessary functions
ic_func3.c: application dependent decoding functions
ic_state.c: functions to save and restore states of ic's.
actions.c: action functions

Input Files

*.in: f, g functions of index cell
fuzzy.dat: (not to be modified): fuzzy computation table.
ic.dat: IC specification for IC Compiler
*.tpl: template input file to IC Compiler (user can modify it)

Manuals
f.g format: f, g function format
output_ic_msg: the usage of "output ic" and "output msg"
in fg function format
ic_prog: manual for ic programming
msg.format: FAKE external message format for the testing of
your active index system

5. TAOML

This directory contains the TAOML interpreter. Makefile.intercgi will generate an executable "inter.cgi". Makefile.inter will generate an executable command "inter".

6. MICE APPLICATION DEVELOPMENT STEPS

The MICE Application Development Steps are as follows:

Step 1. Download IC_Builder to PC and use it to create *.in, *.gra and ic.dat files.

Step 2. Upload *.in and ic.dat to your working directory. Create two sub-directories called "source" and "TAOML". Leave *.in in this directory, and move ic.dat to "source" directory.

Step 3. Change to source directory and do the following:
Step 3.1. Copy all the files from the three
directories IC_Compiler, IC_Manager and IC_Taoml.
to this "source" directory.
Step 3.2. Move ic.dat and action*.c files into the source directory.
Step 3.3. Invoke IC Compiler by typing:
icc ic.dat
Step 3.4. Use the makefiles to make main.cgi and inter.cgi:
make makefile.maincgi
make makefile.intercgi
Step 3.5. Move main.cgi and inter.cgi programs to parent directory.
Step 4. Create index.html that is the home page of your application.
It should invoke inter.cgi to go to another taoml page.
Step 5. Change to TAOML directory and do the following.
Step 5.1. create *.taoml pages which should invoke main.cgi to activate
ic's.
Step 5.2. create *.tpl pages which should invoke inter.cgi to access

Another taoml page, or invoke customized cgi to do special processing.

Step 6. You are now ready to test the application. Use a web browser to access your application's home page index.html.

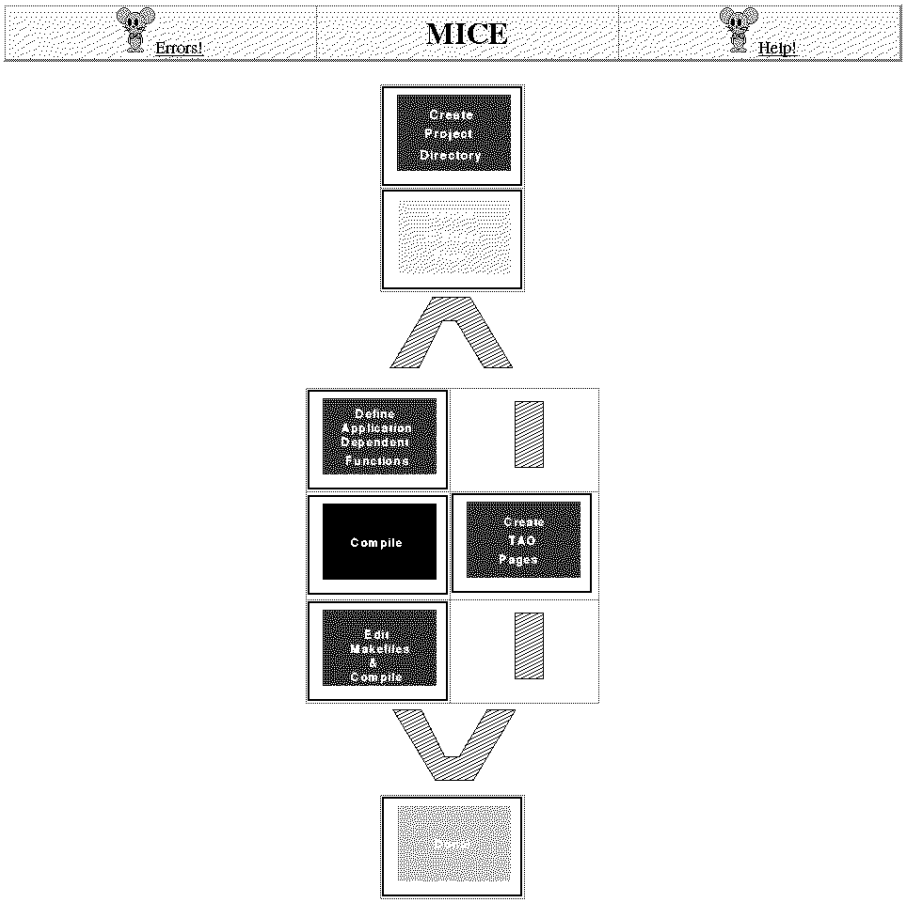

Figure 10. A visual diagram for MICE.

7. VISUAL INTERFACE FOR MICE

The visual interface for MICE is intended for the end user, so that the user does not have to memorize the development steps and the details of multimedia application development using MICE. As illustrated in Figure 10, VISUAL MICE provides a visual diagram. All the user has to do is to

follow the visual diagram and provide the appropriate information at each step.

8. MICE APPLICATIONS

The MICE design environment can be applied to designing all kinds of active multimedia information systems. In what follows, we describe a recent application to active medical information system design [Chang98a, Chang98b].

To accomplish the retrieval, discovery and fusion of medical information from diverse sources, an active medical information system capable of retrieving, processing and filtering medical information, checking for semantic consistency, and structuring the relevant information for distribution is needed. We have developed a framework for the human- and system-directed retrieval, discovery and fusion of medical information, which is based upon the observation that a significant event often manifests itself in different media over time. Therefore if we can index such manifestations and dynamically link them, then we can check for consistency and discover important and relevant medical information.

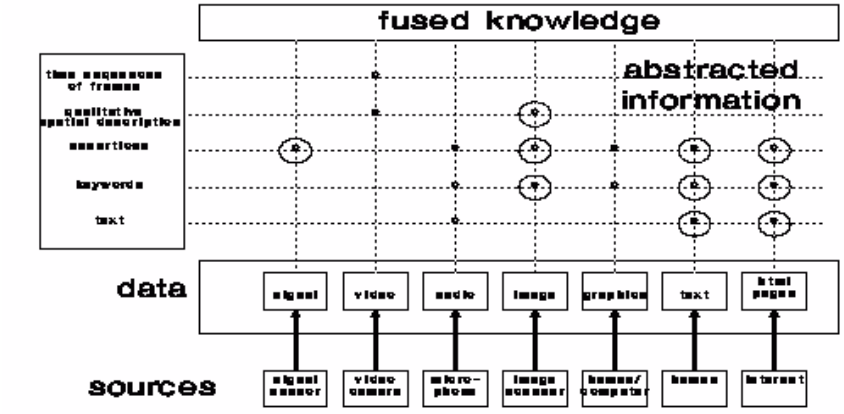

Figure 11. A framework for information and knowledge fusion.

This dynamic indexing technique is based upon the theory of active index. A powerful newly developed artificial neural network is used for the discovery of significant events. An experimental system was implemented, and MICE was used as the prototyping environment to prototype AMIS.

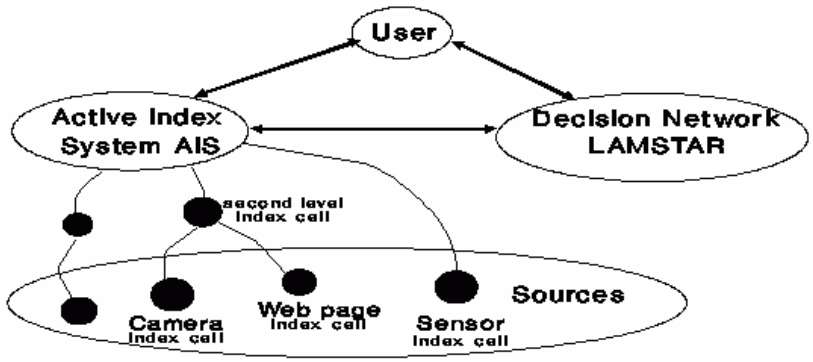

Figure 12. An active medical information system AMIS.

We will give an example to illustrate information fusion by horizontal/vertical reasoning. Patient information is abstracted from different media sources, including imaging devices, signal generators, instruments, etc. (vertical reasoning). Once abstracted and uniformly represented, the neural network is invoked to make a tentative diagnosis (horizontal reasoning). Using the active index, similar patient records are found by the Recursive Searcher (vertical reasoning). A retrieved patient record is compared with the target patient record (horizontal reasoning). If similar records lead to similar diagnosis then the results are consistent and the patient record (with diagnosis) is accepted and integrated into the knowledge base. If the diagnosis is different then the results are inconsistent and the negative feedback can also help the decision network learn.

In the vertical reasoning phase, in addition to comparing patient data, we can also compare images to determine whether we have found similar patient records. Therefore, content-based image similarity retrieval becomes a part of the vertical reasoning. Depending upon the application domain, image similarity can be based upon shape, color, volume or other attributes of an object, spatial relationship among objects, and so on.

This example illustrates the alternating application of horizontal reasoning (using the LAMSTAR neural network for making predictions) and vertical reasoning (using dynamically created active index for making associations). Combined, we have an active information system for medical information fusion and consistency checking.

A demo of the active medical information system can be found at: *http://www.cs.pitt.edu/~chang* and then click on Active Medical Information Systems.

MICE has also been used to prototype an emergency management system [Khali96], an intelligent multimedia information retrieval system [Catar98], a multimedia distance learning system [Chang98c] and other multimedia applications.