

OREGON STATE

UNIVERSITY

COMPUTER

SCIENCE

DEPARTMENT

The SS/1 Design Editor
A Graphical Interface for Object Oriented Parallel Programming
with Server System/1

Thomas Sturtevant
Ted G. Lewis
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

90-60-15

The SS/1 Design Editor
A Graphical Interface for Object Oriented Parallel
Programming
with Server System/1

by
Thomas Sturtevant

A research project submitted in partial fulfillment of
the degree of Master of Science

Major Professor: Ted G. Lewis

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

April 1990

Table of Contents

1. Overview	1
2. Object Oriented Parallel Programming	2
2.1 Object Oriented Parallel Programming with SS/1	2
3. Other Object Oriented Parallel Programming Languages	5
4. The Server System/1 Design Editor	6
4.1 Functional Capabilities	6
4.1.1 Common Features	6
4.1.2 Class Definition	8
4.1.3 Communication Specification	9
5. Design Example: Computer Store Simulation	10
5.1 Computer Store Class Definition	10
5.2 Computer Store Simulation Communication Specification	11
6. Implementation of the SS/1 Design Editor	15
6.1 Implementation Difficulties	15
6.2 Program Statistics	15
7. Practical Use of the SS/1 Design Editor	15
7.1 Presenting Information	15
7.2 Storing Information	16
8. Future Work	16
8.1 Additional Language Constructs	16
8.2 Constant Definitions	17
8.3 Data Flow Analysis	17
Appendix A: Menu Reference for the SS/1 Design Editor	18
Appendix B: SS/1 Design Editor Tutorial "Matrix Multiplication"	23
Appendix C: SML Specification	38
References	39

Abstract

The Server System/1 (SS/1) Design Editor is a graphical programming tool that provides a simple interface for object oriented parallel programming. Two difficult problems with parallel programming, partitioning, and visualization are addressed by the SS/1 Design Editor. An SS/1 program is structured using object oriented design concepts. This leads to highly cohesive program units, which are ideally suited for control-level parallelism. The program is displayed as a graph that explicitly shows the parallel sequencing and temporal dependencies of the messages. Because SS/1 supports some of the basic features of object oriented programming, inheritance and code reuse are encouraged.

1. Overview

Major software performance improvements can be achieved through parallel processing. This performance gain is usually accompanied by increased programming effort to produce parallel code. Whereas textual programming languages are well suited for sequential code, this type of program description is not expressive enough to adequately describe complex parallel sequencing. A graphical programming language can show the parallel structure and temporal dependencies in a program more clearly than a text description.

Communication overhead between processes can be a serious problem for parallel programs. One of the many benefits of object oriented design is low coupling between program units. Because this minimizes communication between the units, it makes object oriented design an ideal method for partitioning a program for control-level parallel processing.

The Server System/1 (SS/1) Design Editor is a graphical, object oriented parallel programming tool. SS/1 allows a programmer to first define a class hierarchy, and then describe a program as a parallel sequence of messages. By displaying the

message sequence graphically in two dimensions, the SS/1 Design Editor makes the task of programming in parallel much easier. Because the resultant program is based on an object oriented design, communication between processes is minimized. The program descriptions generated by the SS/1 Design Editor can be executed using the SS/1 Server on a Sequent Balance parallel computer.

2. Object Oriented Parallel Programming

Both object oriented programming, and parallel programming have been the focus of a tremendous amount of research effort over the past five years. Server System/1 utilizes the benefits of each by allowing messages to be dispatched in parallel. Object oriented programming facilitates encapsulation, information hiding, code reuse, rapid prototyping, and maintainability [Budd]. Parallel processing can lead to dramatic performance improvements by solving different parts of a problem simultaneously. A major problem with parallel programming is communication overhead between the individual processes. To minimize this overhead, it is best to divide the code into highly cohesive program units. In object oriented programming, classes are designed for high cohesion and low coupling to increase encapsulation, and information hiding. Thus an object oriented design is likely to be a good means of partitioning a program for parallel processing.

In an object oriented program, actions are carried out in response to messages. The message indicates what is to be done, but not how or where it should be done. This "message independence" is another feature of object oriented programming which makes it highly compatible with parallel programming.

2.1 Object Oriented Parallel Programming with SS/1

Server System/1 is an object oriented parallel programming system which executes messages as communicating parallel processes [Cho90]. SS/1 requires two inputs: a description of the

class hierarchy, and sequence/dependency information for messages (see appendix B for BNF specification of SML language). Given this information, it finds appropriate methods based on the class hierarchy, and runs them as individual processes according to the sequence guidelines.

The "Server Manipulation Language" (SML) used by SS/1 is a concise, architecture independent description of the class hierarchy and the message scheduling. As a programming language, SML is very clumsy and unmanageable. The SS/1 Design Editor provides a simple, graphical environment for developing SS/1 programs, and automatically generates SML code from the users design. Fig 2.1 shows the interaction between the SS/1 Design Editor, and the SS/1 Server program.

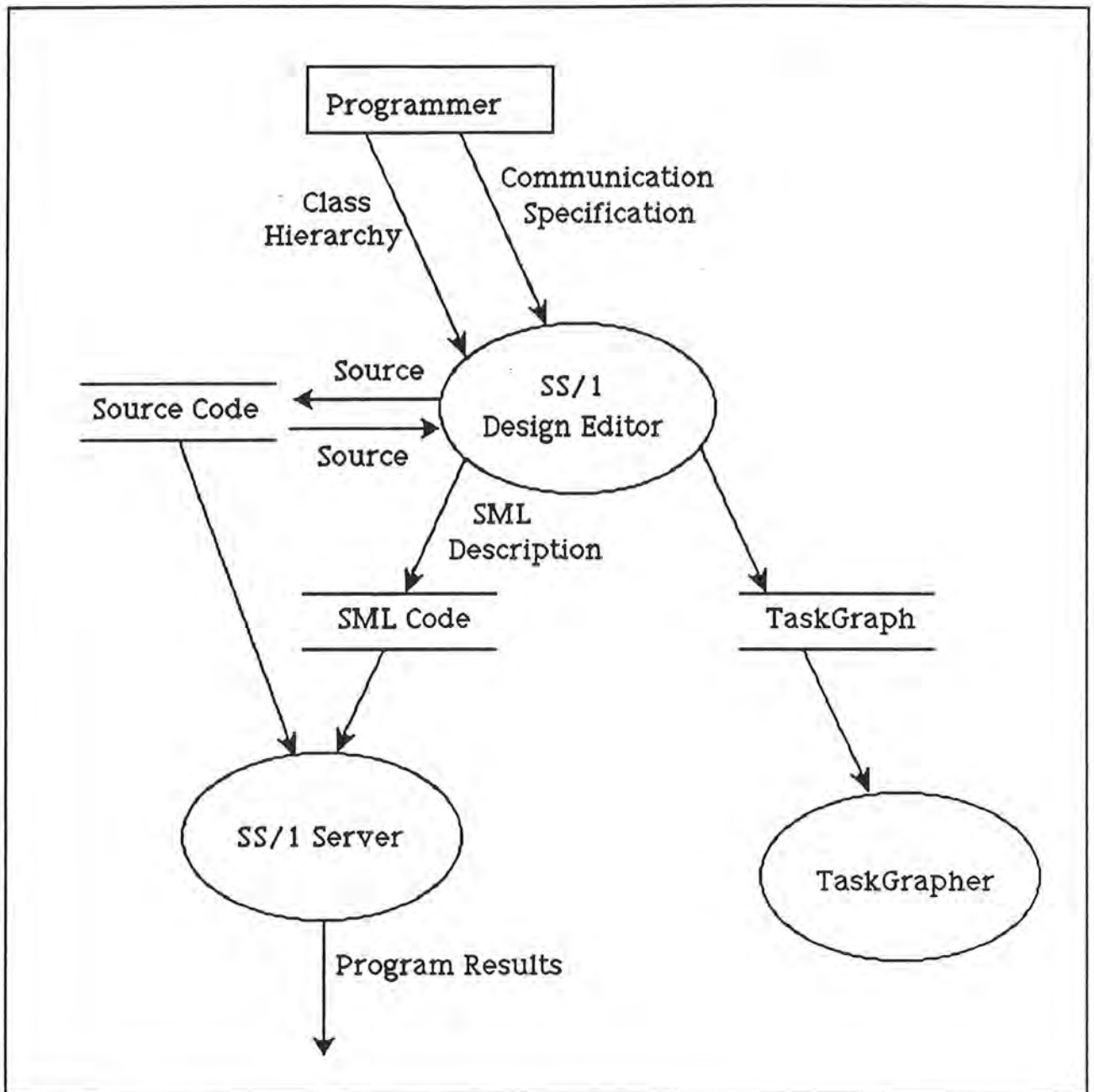


Figure 2.1: SS/1 Data Flow Graph

3. Other Object Oriented Parallel Programming Languages

Because the object oriented approach to large grain parallelisms has many benefits, a number of object oriented parallel programming languages have been developed throughout the 1980's [Yon87]. As in SS/1, many of these languages implement "objects" as independent processing agents which communicate by sending messages.

Act 1 is an implementation of the Actor[Agh87][Lie87] model. In this model objects (called "actors") are separate processes which respond to incoming messages. Each actor maintains a message queue so that messages are handled first-come, first-served. "Future actors" are used as place holders for results being computed. This tends to increase parallelism because an actor must only wait for the future value if a read operation is requested before the value is computed.

ABCL/1[Yon86] is similar to Act 1, but contains some extra features. Messages can be sent in "normal" or "express" mode. An express message will interrupt an object handling a normal message. Message passing can be any of three types: "past" type is asynchronous, "now" type is synchronous, and "future" type is asynchronous and creates a future object for the return value. As in SS/1, the actual methods in ABCL/1 can be written in various languages including C, FORTRAN, LISP, etc.

ConcurrentSmalltalk[Yok86] is a super set of Smalltalk-80 which expands the message passing semantics to include asynchronous (in addition to the standard synchronous) message passing. Results from asynchronous messages are returned to "future objects", similar to the future actors used in Act 1.

Linda Smalltalk[Chun] is an implementation of Little Smalltalk[Bud87] using Linda tuple-space communication. Implicate parallelism is achieved through concurrent execution of message arguments.

Unlike SS/1, none of the afore mentioned languages has a visual programming environment. Parallax[Elr88] is a parallel programming system with a graphical editor similar to that used by

SS/1. However, Parallax programming is not based on object oriented design.

4. The Server System/1 Design Editor

Program generation with the SS/1 Design Editor proceeds in two stages. In the "Class Definition" stage, the user defines a class hierarchy graphically as a tree. This is a natural way to view the hierarchy, and makes the inheritance patterns easy to see (Fig 4.2). In the "Communication Specification" stage a program is defined as a parallel sequence of the methods defined in the Class Definition. This program sequence is displayed as a graph where arcs indicate temporal dependencies between message executions (Fig 4.3). The dependency or "sequencing" graph shows the parallel structure in a very intuitive format. For a programmer this graph is much easier to interpret than the equivalent text description.

4.1 Functional Capabilities

The SS/1 Design Editor closely follows the Macintosh standard user interface guidelines. It provides intuitive, direct manipulation graphical editing of the class definition, and communication specification diagrams. This section describes the basic capabilities of the Design Editor, Appendix B gives a detailed account of all of its features and how they are used.

4.1.1 Common Features

There are many similarities between the class definition, and the communication specification editing capabilities. Both diagrams are displayed in scrolling Macintosh windows. A palette of tools is provided for each diagram (Fig 4.1). The tools available at any time match the type of window which is currently active. Diagrams can be viewed in four discrete "zoom states". This gives the user an option of viewing with high detail, or with a large range. Objects

are selected and manipulated with the mouse. In general, all graphical objects can be selected, moved, and deleted. Groups of objects (with some restrictions) can be represented by a single "compound" object. The contents of a compound object can be viewed in another window, or can be reintegrated into the main diagram. Each type of diagram can be printed, saved and restored by use of an ASCII file.

Class Definition Palette

Communication Specification Palette

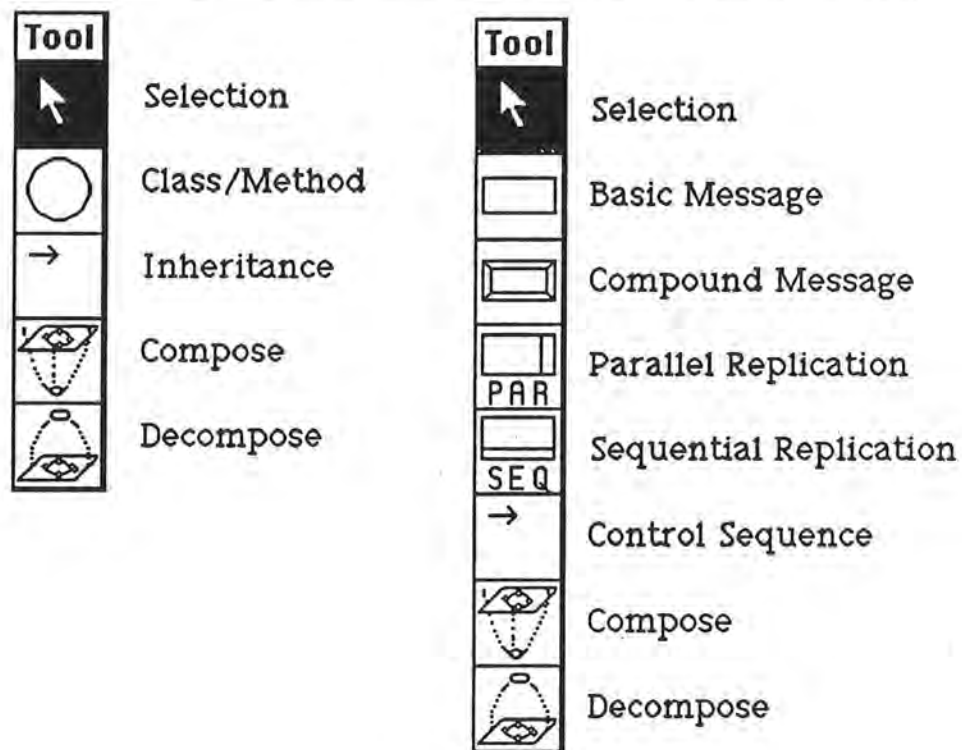


Figure 4.1: SS/1 Design Editor Tool Palettes

concurrently. Finally, a rectangle with a horizontal bar is a compound, sequential replicated message. The messages it represents must be replicated sequentially. Compound messages may have only a single incoming arc, and a single outgoing arc. This single-entry, single-exit restriction removes any chance of ambiguity over the message sequencing.

Information for each message in the diagram must be supplied by the user. For basic messages, the user must specify the receiving class, the method name, and any parameters. The number and types of the parameters will be checked against the method's parameter list in the source file. For compound messages, the user must specify a display label which will be useful for identifying the contents of the message. For replicated messages the user must provide a display label, and the loop bounds.

The communication specification can be used to generate a TaskGrapher[For90] input file. Using TaskGrapher, the programmer can estimate processor scheduling, runtime speedup, and critical path.

5. Design Example: Computer Store Simulation

The Computer Store simulation is an example program written with the SS/1 Design Editor. It simulates the interactions between customers, sales, and service personnel.

5.1 Computer Store Class Definition

The class hierarchy for the Computer Store simulation contains three distinct classes: Ware, People, and Environment. The Class Ware includes all products carried by the store and has subclasses Software and Hardware. The classes Customer, Sales, and Service are all subclasses of class people. The Classes each have methods associated with them as shown in Fig 5.1. Note that the subtrees for classes Hardware and Software are nested within double-bubbles.

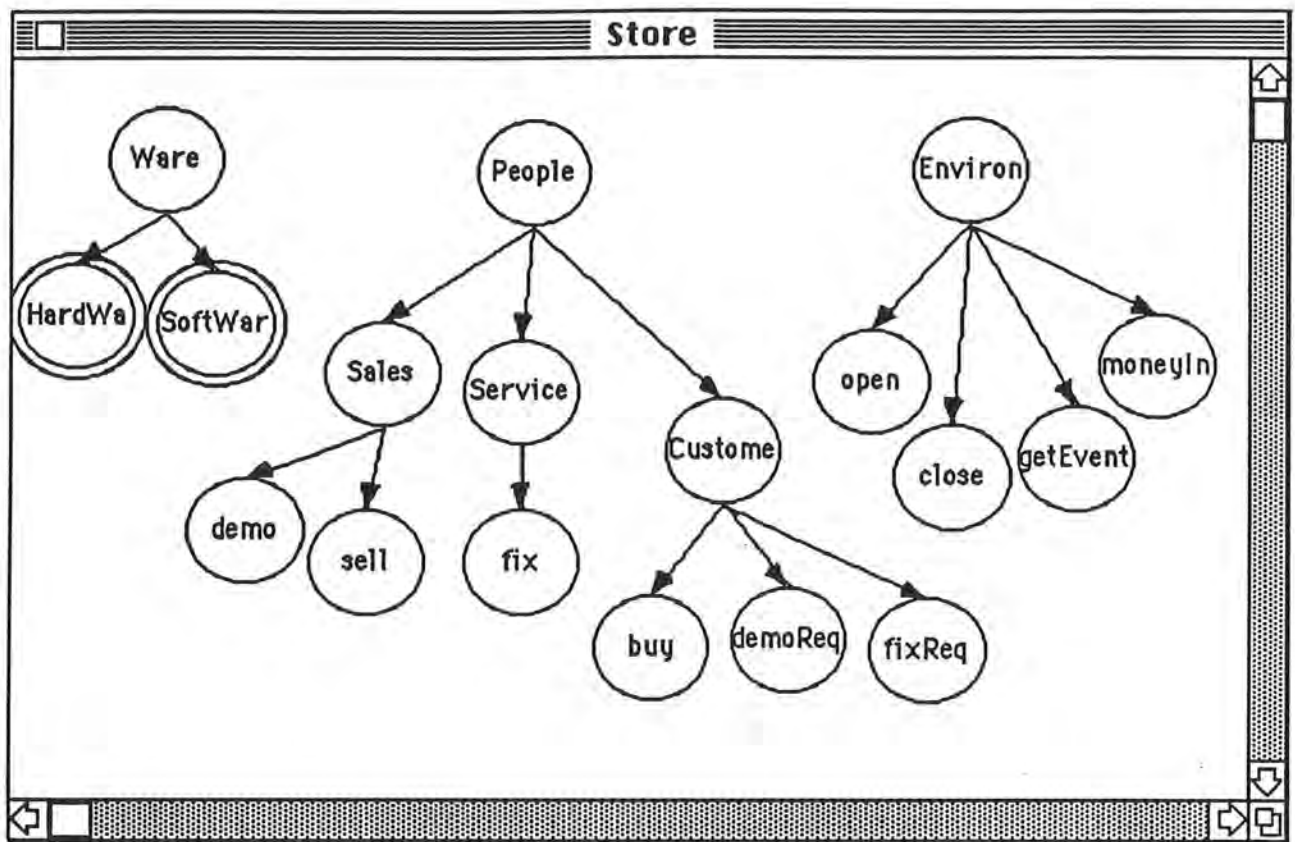


Figure 5.1: Class Definition for Computer Store simulation

5.2 Computer Store Simulation Communication Specification

The top level for the Computer Store simulation communication specification is shown in Fig 5.2. This diagram should be interpreted as follows:

- First open the store.
- Next, customers, sales and service personnel do their actions in parallel.
- Finally, close the store.

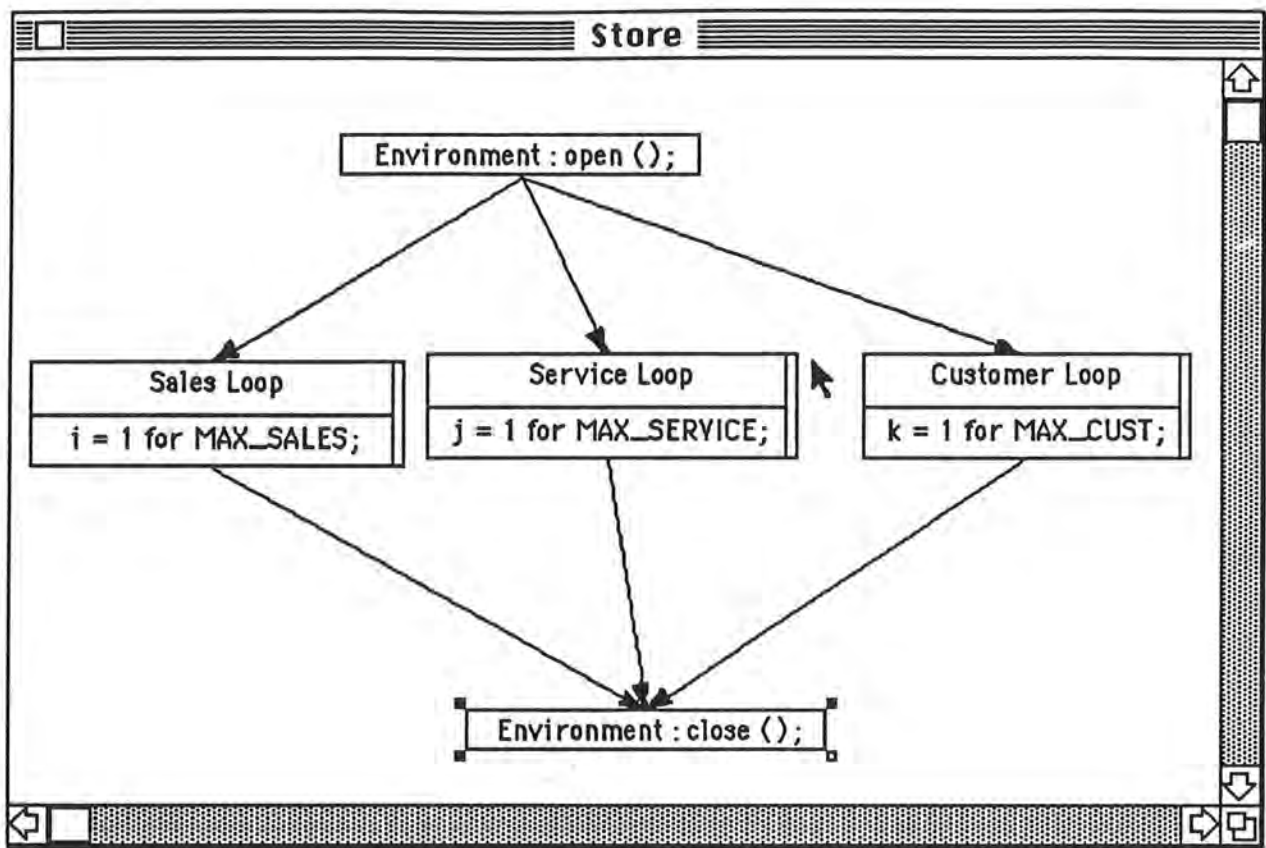


Figure 5.2: Computer Store simulation Communication Specification, Level 1

Nested within each parallel loop is a sequential event loop. Each person in the simulation is generating and/or processing events independent of all others. The loop for an individual sales person is shown in Fig 5.3. This is interpreted as:

- Get an event from the system.
- Depending on the event give a demo, or sell some ware.

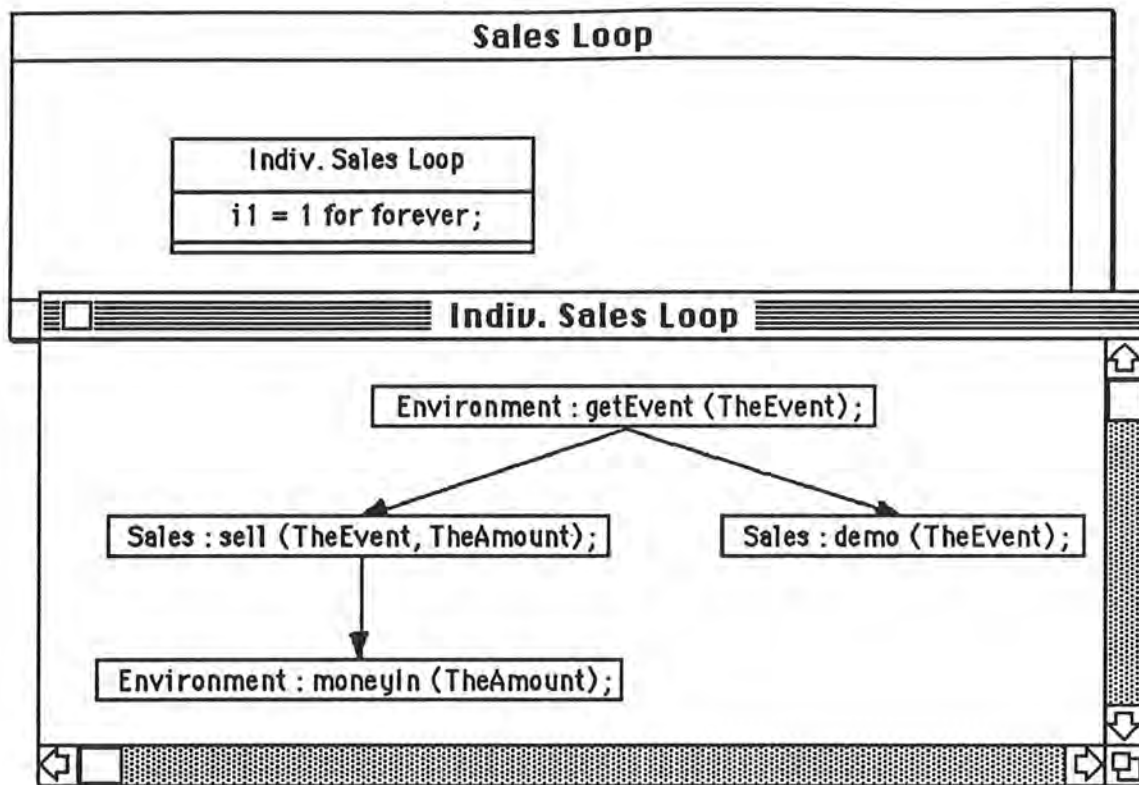


Figure 5.3: Computer Store simulation Communication Specification, Levels 2 and 3

Since SS/1 does not yet support a conditional branch, the demo and sell messages will both be sent. It is thus the responsibility of the demo and sell methods to make sure that theEvent was meant for them. The SML description for this program will be:

```

/* Class Definition */
(
  Ware ( HardWare SoftWare )
  People (
    Sales ( demo sell )
    Service ( fix )
    Customer ( buy demoReq fixReq )
  )
  Environment ( open close getEvent moneyIn )
)
  
```



```

/* Communication Specification */
(SEQ
  (Environment, open)
  (SEQ
    (PAR
      (PAR k = 1 for MAX_CUST
        (SEQ k1 = 1 for forever
          (SEQ
            (Environment, (getEvent, TheEvent))
            (PAR
              (Customer, (fixReq, TheEvent))
              (Customer, (demoReq, TheEvent))
              (Customer, (buy, TheEvent))
            )
          )
        )
      )
    (PAR j = 1 for MAX_SERVICE
      (SEQ j1 = 1 for forever
        (SEQ
          (Environment, (getEvent, TheEvent))
          (Service, (fix, TheEvent))
        )
      )
    )
    (PAR i = 1 for MAX_SALES
      (SEQ i1 = 1 for forever
        (SEQ
          (Environment, (getEvent, TheEvent))
          (PAR
            (Sales, (demo, TheEvent))
            (SEQ
              (Sales, (sell, TheEvent, TheAmount))
              (Environment, (moneyIn, TheAmount))
            )
          )
        )
      )
    )
  )
  (Environment, close)
)

```


6. Implementation of The SS/1 Design Editor

6.1 Implementation Difficulties

The SS/1 design editor was an interesting programming project because it involved implementing a very diverse set of functionality. Major program segments include: two graphical editors, a text editor, a parser for C source code, and a code generator.

One extremely difficult algorithm was the conversion of a multi-layered DAG (i.e. the communication specification diagram) into the corresponding textual SML description.

6.2 Program Statistics

The SS/1 design editor was developed using Think's LightSpeed Pascal 2.0. The Pascal project contains 23 units with a total of nearly 13,000 lines of source code. The compiled size of the application is 101K.

7. Practical Use of The SS/1 Design Editor

Practical (commercial quality) programming environments must be capable of handling very large programs. The SS/1 Design Editor was implemented as a research system, and is not designed to cope with the development of large programs.

7.1 Presenting Information

The SS/1 Design Editor's visual interface presents a program in a simple, useful manner. Abstraction mechanisms (double bubbles and double rectangles) can be used to reduce screen clutter and organize a design in a modular fashion. As a design gets more complex, thoughtful organization becomes more important and at the same time much more difficult. The SS/1 Design Editor does not

provide any assistance with program organization. A program containing many control arcs may become very difficult to trace.

A loop construct is displayed as a double rectangle. The loop body must be viewed in a separate window even if it only consists of a single message. For practical purposes this is too restrictive. To view a complete matrix multiplication program (with a triple nested loop) requires four windows.

7.2 Storing Information

SS/1 Design Editor designs are stored in monolithic ASCII files. The ASCII format is human readable, and makes the design information easily available to other applications. However, large designs require a considerable amount of time for loading and saving. Storing the data in a binary format would speed up this process considerably. The monolithic nature of a design file makes it difficult to coordinate changes made by a team of programmers. All individual changes would have to be merged into the single design file. One natural solution to this problem is to have a separate file to describe each window, but here the problem is likely to be too many files because a simple design can require many windows.

8. Future Work

There are a few additions to Server System/1 and the SS/1 Design Editor that would need to be made in order for it to be a viable, practical tool. The most important of these are listed in this section.

8.1 Additional Language Constructs

SS/1 does not currently support a conditional branch in the SML language. This limitation is reflected in the SS/1 design editor.

A conditional branch is vital in any practical system, and should be added to SS/1.

8.2 Constant Definitions

The SS/1 design editor should provide a method for the user to define constant values. Constants can be included in an SML file, so this is purely an interface issue.

8.3 Dataflow Analysis

Dataflow information contained in message arguments could be utilized to detect data-independent sections of a program which could be run in parallel. Currently the process of identifying parallelism must be handled by the programmer.

Appendix A: Menu Reference for the SS/1 Design Editor

This appendix gives an overview of the SS/1 design editor menus. The usage and availability of each menu item is listed.

Apple Menu



About SS/1

This displays a dialog detailing the date, version, and author of the application.

Desk Accessories

SS/1 supports all desk accessories.



Session Menu

Generate Glue Code

Generate the SML description of the class definition and communication specification diagrams. The user must input the name for the SML file.

Print

Print the current active window. Text and graphics windows may both be printed in this manner.

Quit

Terminate the application. If there are any unsaved changes, the user is given an opportunity to save the changes.

Class Definition	
New	
Open	
Close	SW
Save	
Save As...	

Class Definition Menu

New

Open a new class definition diagram. This menu item is unavailable if a class definition diagram is currently open.

Open

Open an existing class definition diagram. This menu item is unavailable if a class definition diagram is currently open.

Close

Close the current class definition diagram. This menu item is unavailable if no class definition diagram is currently open.

Save

Save the current class definition diagram as an ascii file. This menu item is unavailable if no class definition diagram is currently open.

Save As...

Save the current class definition diagram under a name entered by the user. This menu item is unavailable if no class definition diagram is currently open.

Communication Spec.	
New	
Open	
Close	Alt+W
Save	
Save As...	
Export Task Graph As...	

Communication Specification Menu

New

Open a new communication specification diagram. This menu item is unavailable if a communication specification diagram is currently open.

Open

Open an existing communication specification diagram. This menu item is unavailable if a communication specification diagram is currently open.

Close

Close the current communication specification diagram. This menu item is unavailable if no communication specification diagram is currently open.

Save

Save the current communication specification diagram as an ascii file. This menu item is unavailable if no communication specification diagram is currently open.

Save As...

Save the current communication specification diagram under a name entered by the user. This menu item is unavailable if no communication specification diagram is currently open.

Export Task Graph As...

Generate a TaskGraph format file from the communication specification diagram. This menu item is unavailable if no communication specification diagram is currently open.

Edit Text	
Cut	⌘H
Copy	⌘C
Paste	⌘V
Save Text	
Save Text As...	

Edit Text Menu

The Edit Text menu contains the word processing commands for editing method source files. The source files are opened through the Method Info Dialog (discussed in Appendix B section 1.3).

Cut

Delete the currently selected text section, and place it into the paste buffer. This menu item is unavailable if no text is selected.

Copy

Place a copy of the currently selected text section in the paste buffer. This menu item is unavailable if no text is selected.

Paste

Insert the contents of the paste buffer at the current cursor position. This menu item is unavailable if there is nothing in the paste buffer.

Save Text

Save the contents of the active text window. This menu item is unavailable if a text window is not active.

Save Text As...

Save the contents of the active text window under a name entered by the user. This menu item is unavailable if a text window is not active.

Layout	
Redraw	
Zoom Out	⌘-
Zoom In	⌘+

Layout Menu

Redraw

Redraw the active graphics window. This menu item is unavailable if a graphics window is not active.

Zoom Out

Show a greater range of the active graphics window. This menu item is unavailable if a graphics window is not active, or if the active graphics window is already zoomed out to its greatest range.

Zoom In

Show greater greater detail in the active graphics window. This menu item is unavailable if a graphics window is not active, or if the active graphics window is already zoomed in for maximum detail.

Appendix B: SS/1 Design Editor Tutorial "Matrix Multiplication"

This section gives step by step instructions for using the SS/1 Design Editor to write a matrix multiplication program. It is assumed that the reader has some experience using Macintosh applications. Detailed instructions for actions such as selecting menu items, and selecting files from a mini-finder dialog will not be provided.

1. Building a Class Definition Diagram

1.1 Building the Class Hierarchy

Launch the SS/1 Design Editor. Double click on the application icon to launch the design editor.

Open a Class Definition diagram. Select item "New" in the Class Definition menu. At this point two windows should appear on your screen. The large window titled "Class Definition" is where the class hierarchy will be displayed. The small window at the left edge of the screen titled "Tool" is the tool palette containing the tools you will use to construct the class hierarchy.

Choose the Class/Method Tool from the tool palette. The Class/Method tool is represented as a circle in the tool palette. When it is selected the cursor will change to a circle whenever it is over the diagram window.

Place a Class/Method bubble in the diagram window. This is done by positioning the cursor near the top of the diagram window and pressing the mouse button once. At this point a circle (representing a class or a method) is displayed in the diagram window (Fig. B.1).

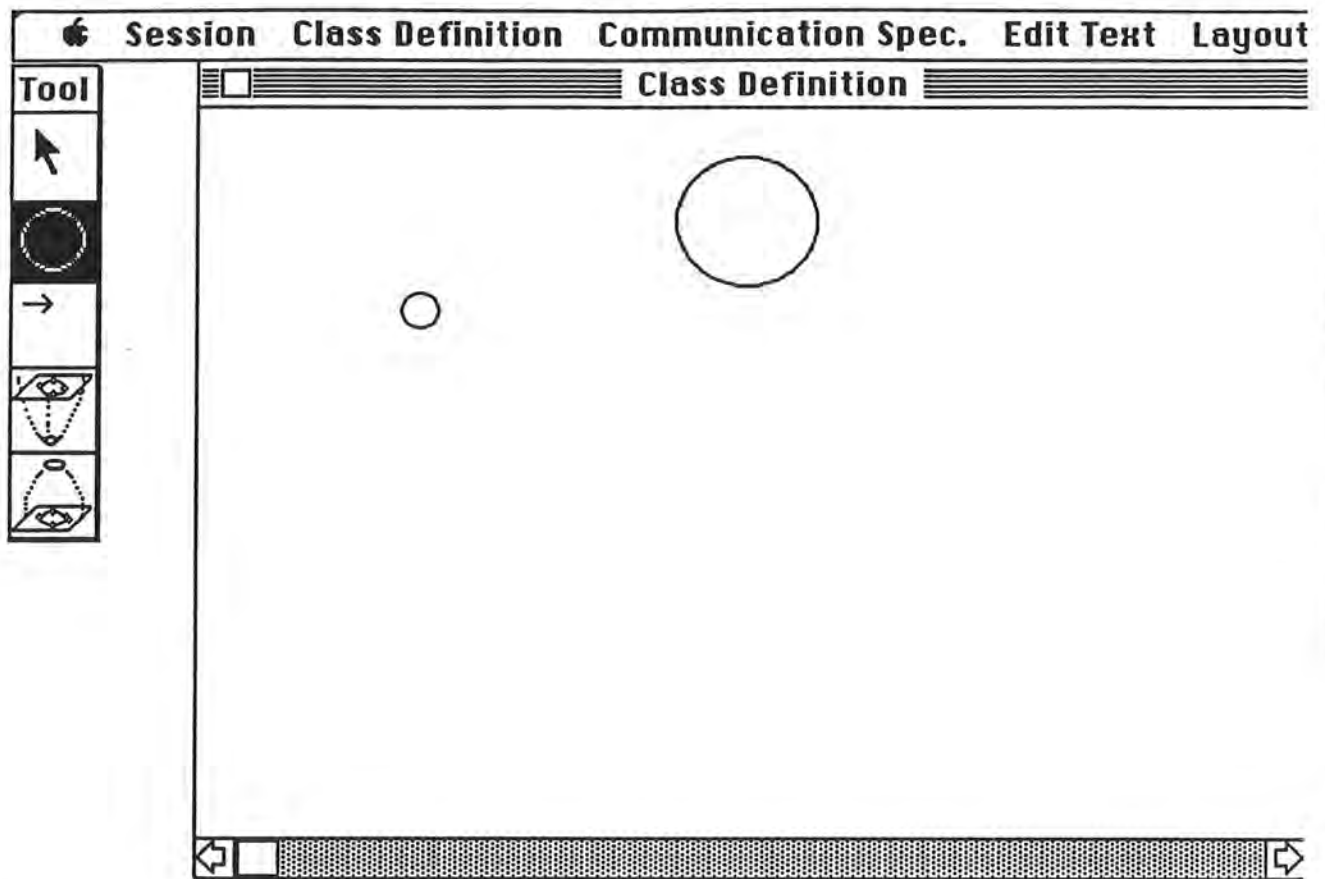
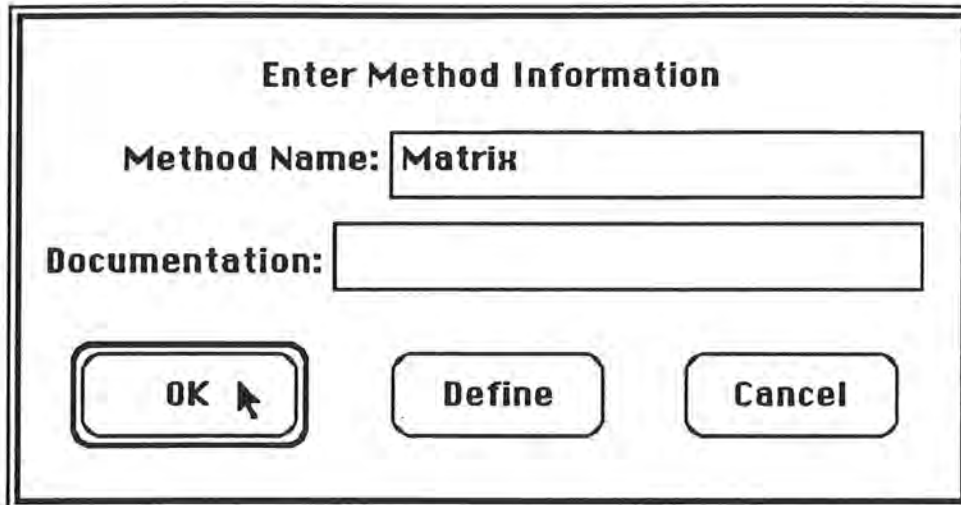


Figure B.1: The Class Definition diagram.

Choose the Selection Tool from the tool palette. The Selection tool is the diagonal arrow at the top of the palette.

Change the name of the new Class to "Matrix". With the selection tool, double click on the circle in the diagram window. A dialog will appear (Fig B.2) with the instructions "Enter Method Information". Type "Matrix" in the field labeled "Method Name" and select the "OK" button. The circle in the window should now be labeled "Matrix".



Enter Method Information

Method Name:

Documentation:

Figure B.2: The Method Information Dialog.

Add "2-D" and "Array" Classes to the diagram. Following the same steps used to create the "Matrix" class, add class bubbles labeled "Array", and "2-D".

Note: our diagram will be neater if the 2-D and Array class bubbles are placed side-by-side and slightly below the Matrix class bubble. Bubbles may be dragged with the Selection tool.

Specify the hierarchy among the classes in the diagram. Choose the Connection tool which is represented by a horizontal arrow in the palette. Click the mouse button down in the Matrix class bubble, and while holding the button down, drag the cursor into the 2-D class bubble and release the button. An arrow should now connect the two classes indicating that 2-D is a subclass of Matrix. Repeat this process so that an arrow also connects classes Matrix and Array (Fig B.3).

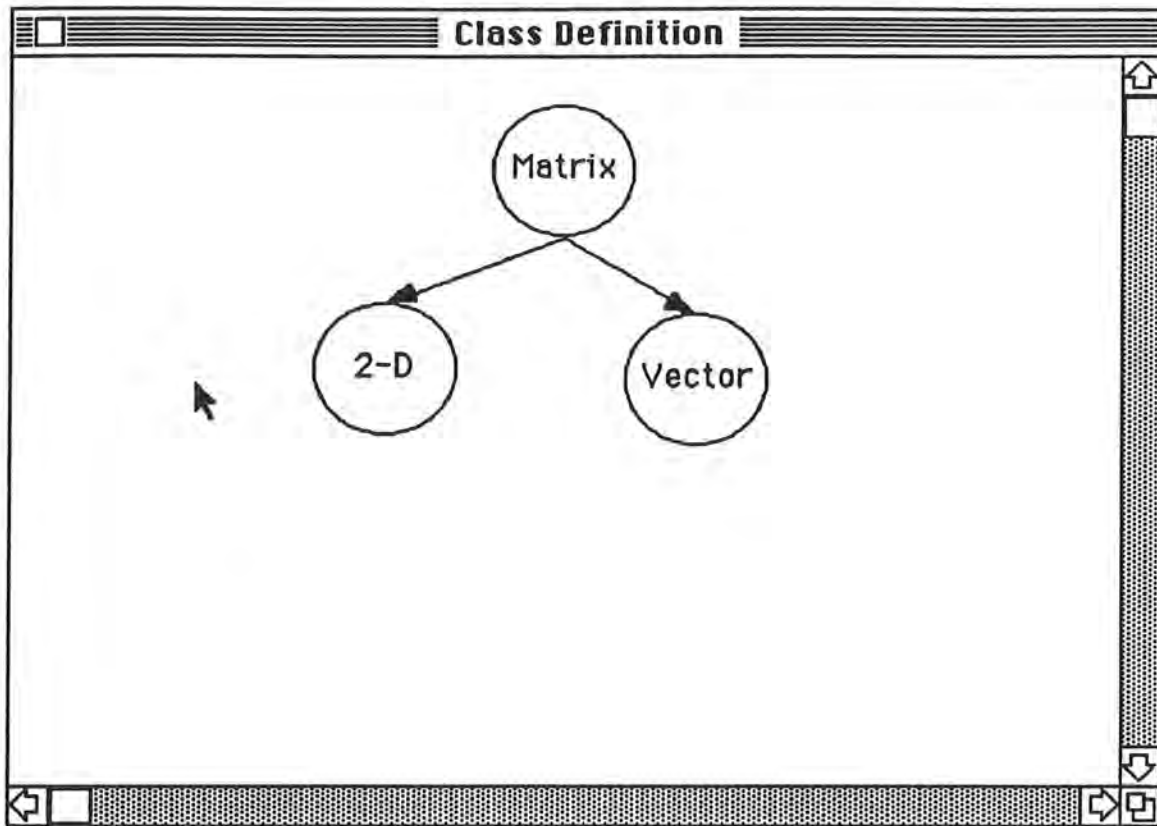


Figure B.3: A Class Hierarchy.

Add Methods "read", "write", and "inner" to class 2-D. Add three bubbles titled "read", "write", and "inner" just below the bubble for class 2-D. Using the Connection tool, add arrows from 2-D to each of the three new bubbles. In the diagram, methods are distinguishable from classes because they are the leaves of the class hierarchy. This does lead to some ambiguity for classes which have no methods defined.

1.2 Some Interface options

Nest class 2-D into a compound bubble. Select the Pack tool, which is next to the bottom in the tool palette. Now select the entire subtree containing 2-D, read, write, and inner by dragging a box around them in the diagram. At this point (if you successfully

selected the whole subtree) a dialog will appear with the instructions "Enter Information for Composed Items:". Type "2-D Mat" in the field labeled "Title" and select the "Define" button. A new window will now appear with the title "2-D Mat" which contains only the four bubbles selected with the Pack tool (Fig B.4).

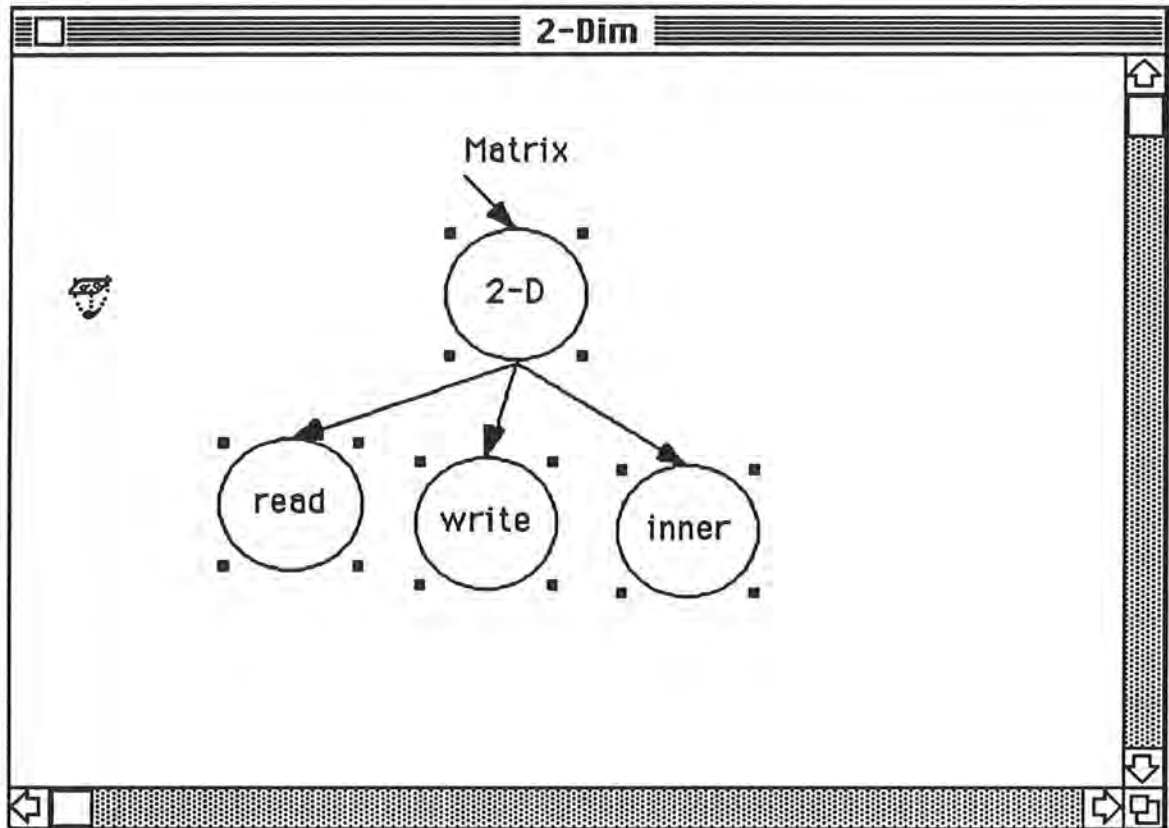


Figure B.4: Lower level of the Class Hierarchy.

UnPack the compound bubble. First close the "2-D Mat" window. This can be done either by selecting the "Close" item in the Class Definition menu, or by clicking the mouse button with the cursor positioned in the close box at the upper left corner of the window. Now the window title "Class Definition" will again be active. Notice that the entire subtree for the 2-D class is now represented by a double-bubble titled "2-D Mat". You may again view this in a separate window by double-clicking on it with the Selection tool,

and selecting "Define" in the resultant dialog. To reintegrate the subtree into the main diagram, choose the UnPack tool at the bottom of the palette and double click on the "2-D Mat" double-bubble.

View the diagram in different zoom states. Select the "Zoom Out" item in the Layout menu. The diagram will now be redrawn smaller. Select the "Zoom In" item in the Layout menu to get the drawing back to its original size. The SS/1 Design Editor provides four discrete zoom states. The diagram can be viewed and/or manipulated in all of the zoom states. By zooming out you lose some resolution, but you can view a larger diagram.

1.3 Source-Level Specification for the Methods.

Open a text window for the read method. Choose the Selection tool, and double click on the bubble for the read method. Select the "Define" button in the resultant dialog. You are now presented with a dialog asking if you want to open a new file, select the "Yes" button. A text window titled "read" will now appear. Enter the text for the matrix read method as in Fig B.5.

The Edit Text menu contains standard "Cut", "Copy", and "Paste" items which you can use to manipulate the text.

Save the read method text. Select item "Save Text As..." from the Edit Text menu, and specify appropriate file and folder for the text file.

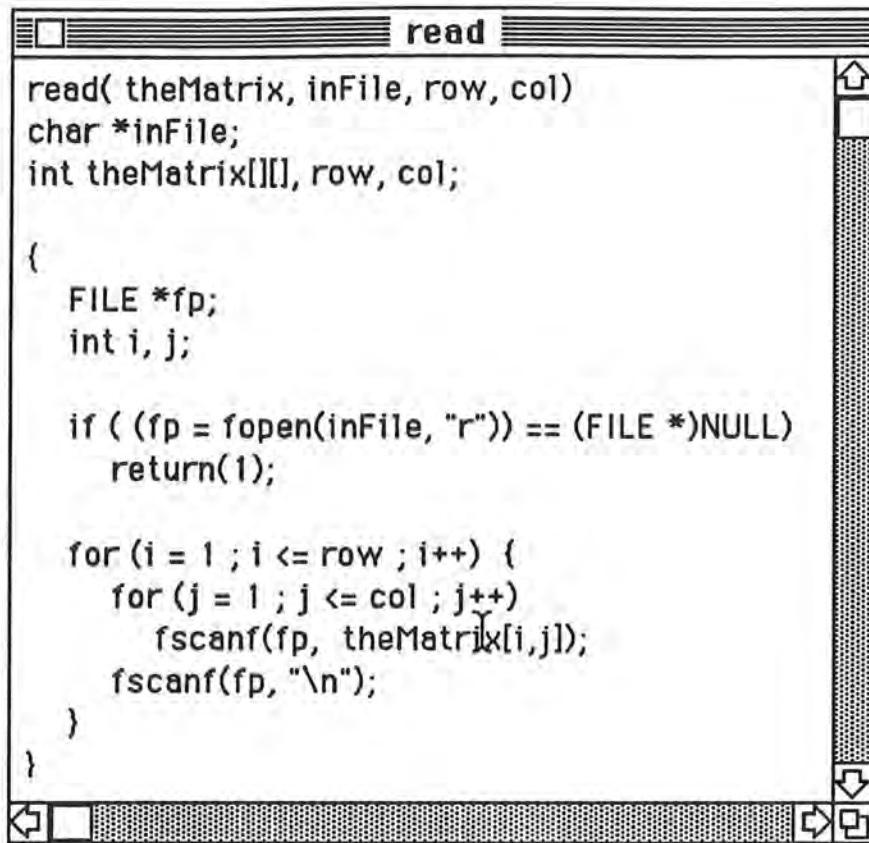


Figure B.5 Text window for method read.

Specifying an existing source file for a method. If source for a given method already exists, you can "link" that source to the appropriate method in the class definition diagram. To do this, double click on a method using the Selection tool, and select the "Define" button, now select the "No" button in the next dialog which inquires whether you want to open a new file. You can now select the appropriate source file for the method with the standard file dialog.

Specify source code for write, and inner methods. Now you know two ways to "link" source code to the methods in the class definition diagram. Use one or both of them to associate the following code with the write, and inner methods:

```

write( theMat, row, col )
float theMat[][];
int row, col;
{ int i, j;
  for (i = 0; i < row; i++) {
    for (j= 0; j < col; j++)
      printf("%d", theMat[i][j]);
    printf("\n");
  }
}

inner( MatA, MatB, MatC, rowCol, row, col )
float MatA[][], MatB[][], MatC[][];
int rowCol, row, col;
{ int i;
  MatC[row][col] = 0.0;
  for (i = 0; i < row; i++)
    MatC[row][col] += MatA[row][i] * MatB[i][col];
}

```

Save the Class Definition diagram. Select item "Save As..." in the Class Definition menu, and specify a file name of "Matrix.classDef". The window title will now change to "Matrix".

The class definition diagram is now complete, so we move to the next phase: The Communication Specification.

2. Building a Communication Specification Diagram

Open a Communication Specification diagram window. Select item "New" in the Communication Spec menu. At this point a window entitled "Communication Spec" will appear, and the tool palette will change and display the tools for editing a communication specification diagram (Fig B.6).

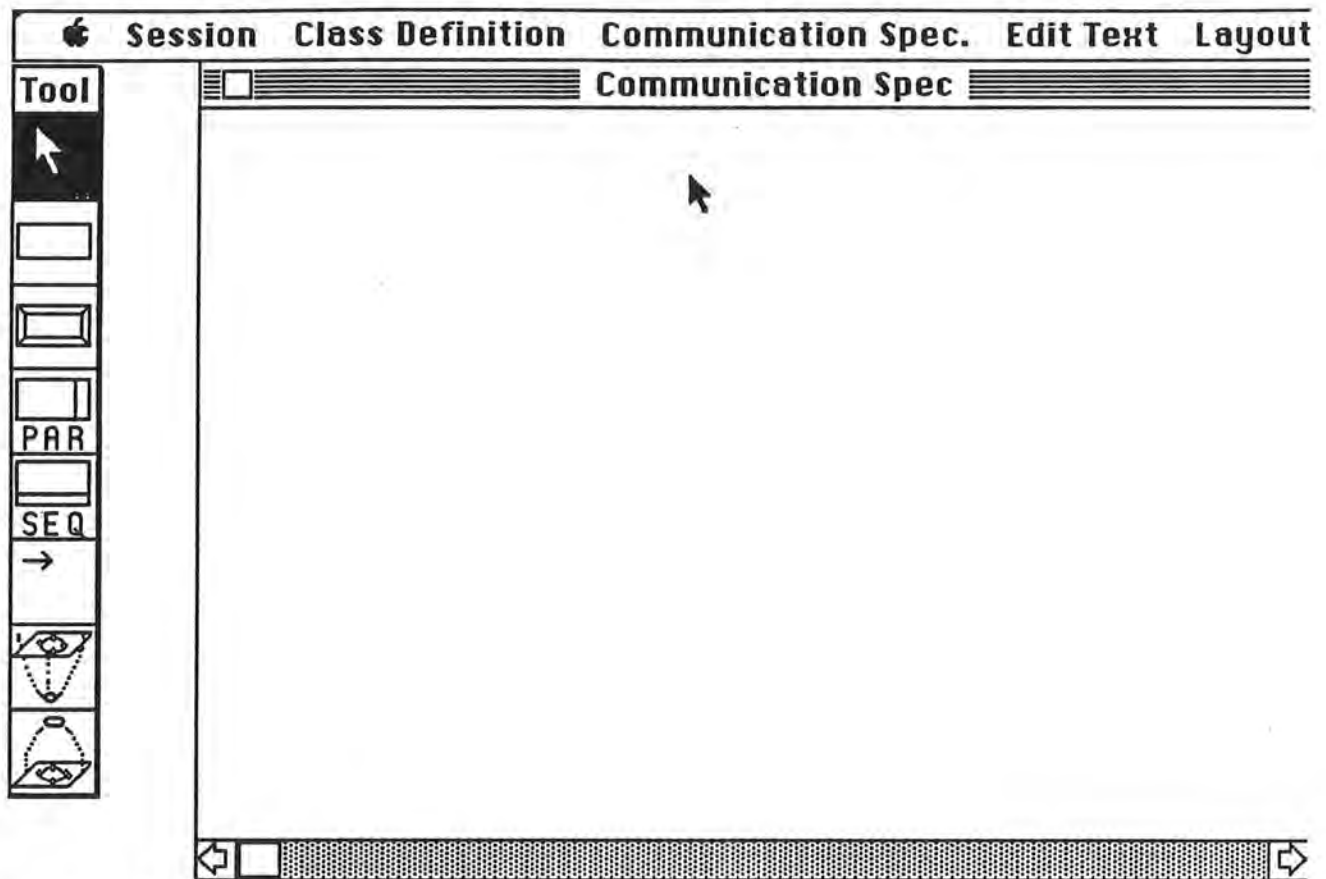


Figure B.6: The Communication Specification window.

Place messages for reading in two matrices. Choose the Basic Message tool, which is a plain rectangle (the second icon in the tool palette). With this tool, place two message rectangles near the top of the communication specification diagram window.

Fill in message parameters. The two messages in the window will be used to read in the two matrices that we want to multiply. We will use the 2-D, read method to accomplish this. Activate the class definition window now titled "Matrix" (notice how the tool palette changes back to the class definition tools), and with the Selection tool select the read method. Now reactivate the Communication Specification window, and with the Selection tool double click one of the basic method rectangles. A dialog will appear with the instructions "Enter Message Information" (Fig B.7). The class, and

method fields will be filled in with "2-D", and "read" respectively (because that is the method currently selected in the class definition diagram). In the "Message Arguments" field enter: 'MatA, "DataA", 40, 50'. And select the "OK" button. At this point the application will check the number and type of the arguments against the parameters declared in the source file.

Enter Message Information:

Class:

Method:

Message Arguments:

Figure B.7: The Message Information Dialog.

Fill in parameters for the other message. For the other basic message specify the following parameters in a similar manner: 'MatB, "DataB", 50, 60'.

Place a parallel replicator message in the diagram. Choose the Parallel Replicator Message tool (labeled PAR in the palette). Place a parallel replicator message below the two read message rectangles in the diagram. This will be displayed as a larger rectangle with a vertical bar on the right side.

Fill in parameters for the parallel replicator. With the Selection tool, double click in the parallel replicator, and fill in the fields as in Fig B.8. Select the "OK" button.

The dialog box has a title bar that reads "Enter Information for Parallel Replicator:". Inside, there are four labeled input fields: "Loop Label:" with the text "Multiply", "Loop Variable:" with the text "i", "Initial Value:" with the text "1", and "Final Value:" with the text "40". At the bottom, there are three buttons: "OK" (which is highlighted with a mouse cursor), "Define", and "Cancel".

Figure B.8: The Parallel Replicator Info Dialog.

Specifying sequence for the three messages. Select the Connection tool (the horizontal arrow). Position the mouse in one of the basic (read) messages, drag to the parallel replicator message and release. Repeat this process to connect the other basic message to the parallel replicator (Fig B.9). The arcs that you have just drawn indicate a sequencing dependency for the program, i.e.: you must read in the two matrices before you can multiply them.

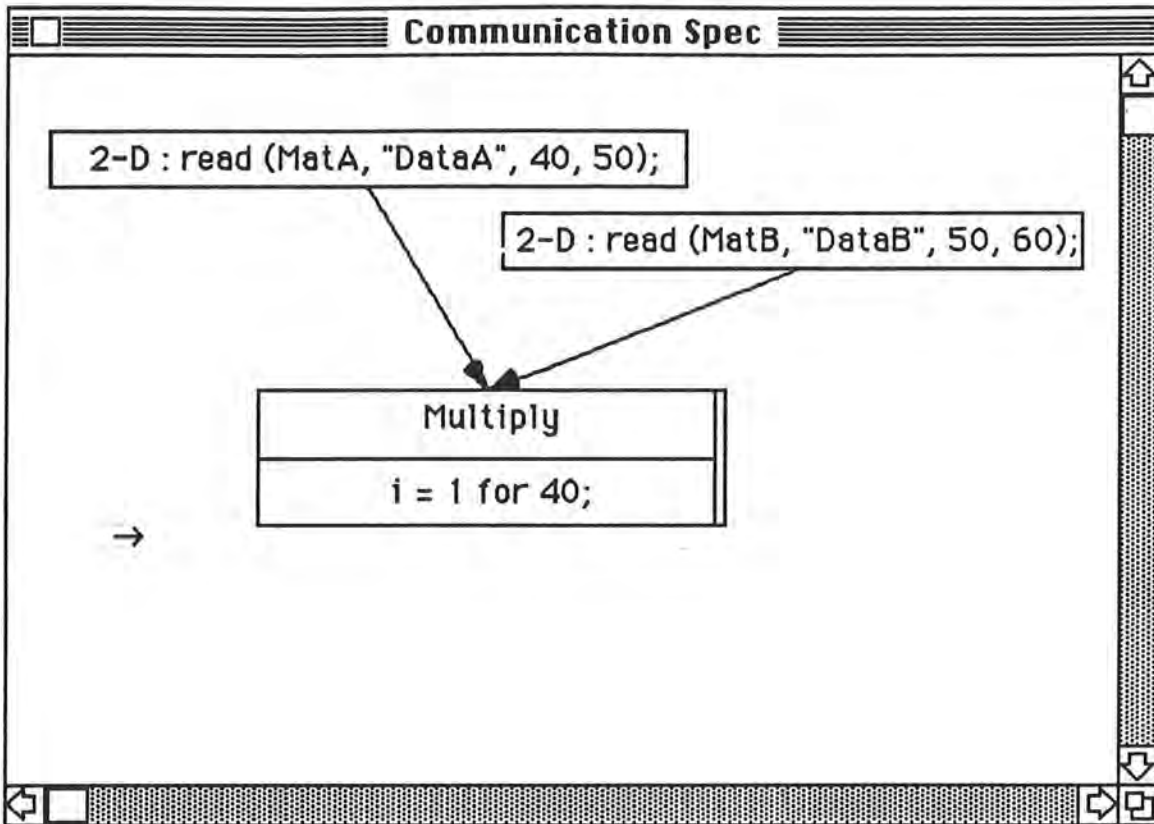


Figure B.9: Message sequencing.

Fill in the second level of the replicator message. With the Selection tool, double click the parallel replicator message, and select the "Define" button in the resultant dialog. At this point a new window titled "Multiply" will appear. In this window you will indicate what messages will be sent within the parallel loop. Place another parallel replicator message in this window (Matrix multiplication is basically a nested loop). For this message fill in the parameters: field Loop Label -> "Inner Loop", field Loop Variable -> "j", field Initial Value -> "1", field Final Value -> "60".

Fill in the lowest level of the multiplication loop. Select "Define" in the dialog for the Inner Loop parallel replicator and another window titled "Inner Loop" will be displayed (Fig B.10). Within this window place a basic message to calculate the inner product. The parameters for this message will be: Class -> "2-D", Method ->

"inner", Message Arguments -> "MatA, MatB, MatC, i, j". Now close the window titled "Inner Loop", and then the window titled "Multiplication".

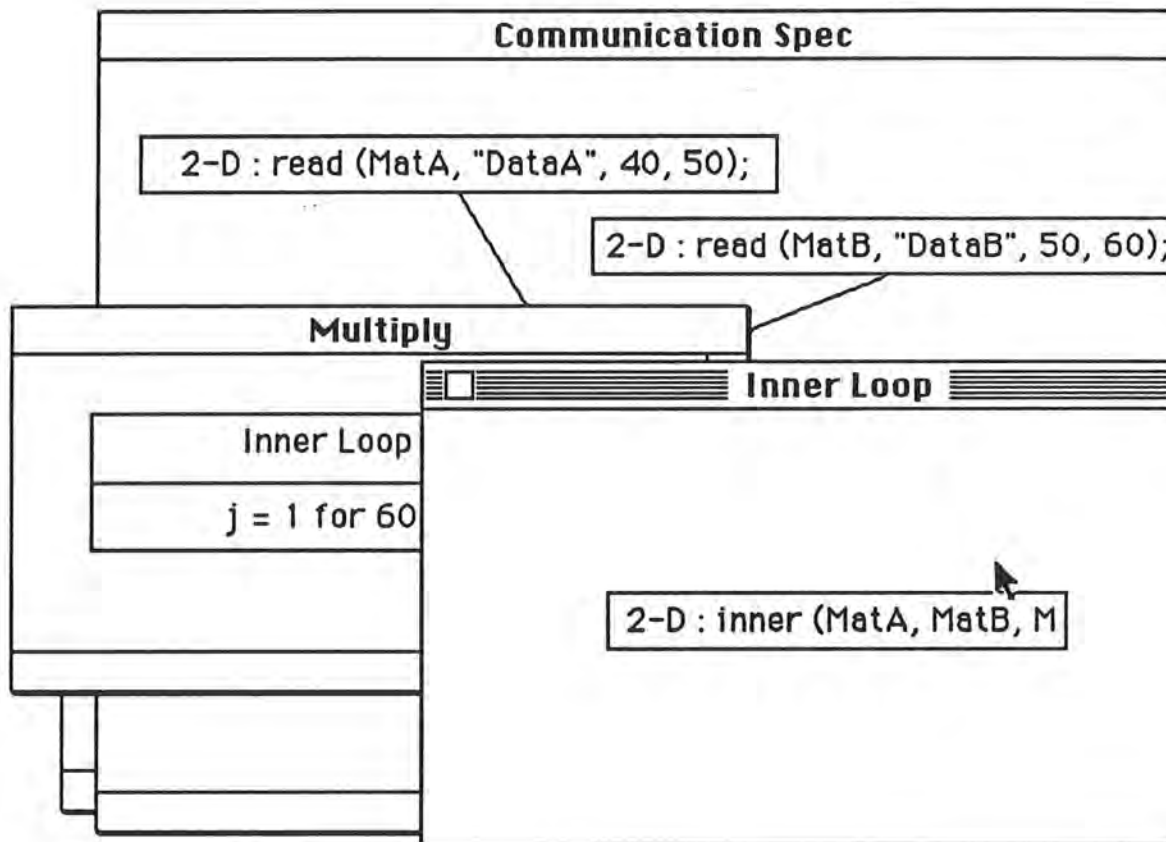


Figure B.10: Nested levels in the Communication Specification.

Place message for writing the resultant matrix. In the top level communication specification diagram place another basic message for writing the resultant matrix. Parameters for this message will be: Class -> "2-D", Method -> "inner", Message Arguments -> "MatC, 40, 60". Now add a sequencing arc to indicate that MatC can be written when the Multiplication loop is completed (Fig B.11).

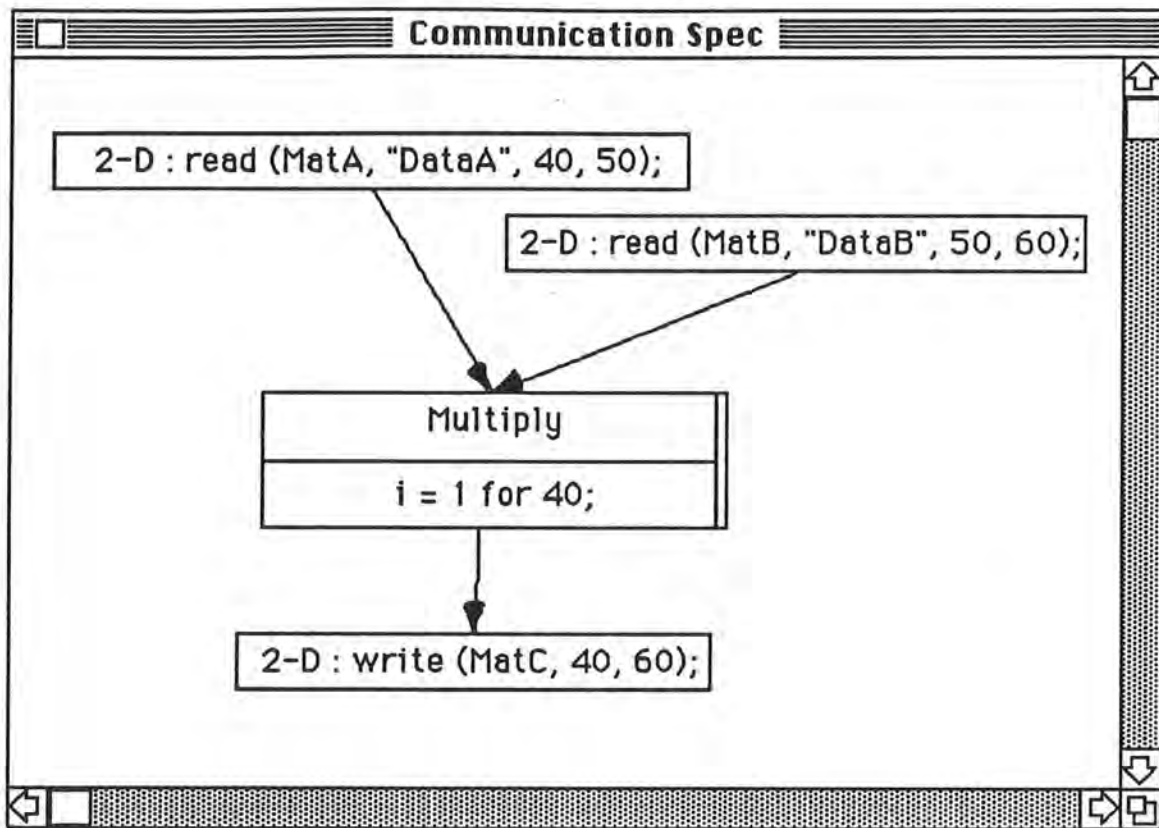


Figure B.11: The complete Communication Specification.

Saving the Communication Specification Diagram. Select item "Save As..." in the Communication Specification menu, and specify a file name of "MatrixMult.commSpec". The window title will now change to "MatrixMult".

3. Generating SML Glue Code and TaskGraph output

Generate the SML glue-code for the program. Select item "Generate Glue Code" in the Session Menu. You will now get a warning that the Method "Vector" was not used. Ignore the warning. Now choose a file name for the SML code. The SML code for your program will look like this:

```

/* Class Definition */
(
  Matrix (
    2-D (
      write
      inner
      read
    )
    Vector
  )
)

/* Communication Specification */
(SEQ
  (PAR
    (2-D, (read,MatB, "DataB", 50, 60))
    (2-D, (read,MatA, "DataA", 40, 50))
  )
  (PAR i = 1 for 40
    (PAR j = 1 for 60
      (2-D, (inner,MatA, MatB, MatC, 50, i, j))
    )
  )
  (2-D, (write,MatC, 40, 60))
)

```

Generate TaskGraph format description of the Communication Specification. Select item "Export Task Graph As..." in the Communication Spec. Menu. Select a name for the TaskGraph format description of the communication specification diagram. You can read this into TaskGraph to study the parallel execution of your program.

Appendix C: SML Specification

```
program          : class_definition com_specifications ;
class_definition : ( class_declarations )
                  | null ;
class_declarations : class_declarations class_declaration
                    | class_declaration ;
class_declaration : class_name ( class_declarations )
                  | class_name
                  | method_name ;
com_specifications : com_specifications messages
                   | messages ;
messages          : ( construct messages )
                   | messages ;
construct         : SEQ
                  | PAR
                  | SEQ replicator
                  | PAR replicator ;
replicator        : var = start to end ;
start             : var
                  | integer ;
end              : var
                  | integer ;
message          : ( class_name, method_dsc ) ;
method_dsc       : ( method_name, arg_list )
                  | method_name ;
arg_list         : arg_list expression
                  | expression ;
expression       : var
                  | integer
                  | string ;
```

References

- [Cho90] Sungwoon Choi, Tom Sturtevant, Ted G. Lewis, *Parallel Programming and Designing in Object Oriented Environment SS/1*, Technical Report, Computer Science Department, Oregon State University, April 1990.
- [Bud87] Timothy A. Budd, *A Little Smalltalk*, Addison-Wesley, 1987
- [Budd] Timothy A. Budd, *An Introduction to Object Oriented Programming*, To appear.
- [Chun] Jean Chung, Timothy A. Budd, *LINDA SMALLTALK: Parallel Implementation of Little Smalltalk Using Tuple Space Communication of Linda*, Technical Report, Computer Science Department, Oregon State University, To appear.
- [Mac86] Apple Computer Inc., *Inside Macintosh*, Volume I, II, III, and IV, March 1986.
- [Bab84] Robert G. Babb, *Parallel Processing with Large-Grain Data Flow Techniques*, IEEE Computer pp. 55-61, July 1984
- [For90] Patrick D. Fortner, Dr. T. G. Lewis, *MacSchedule: Speedup Estimation in Parallel Program Task Graphs*, Technical Report, Computer Science Department, Oregon State University, April 1990.
- [Elr88] Hesham El-Rewini, Ted Lewis, *Software Development in Parallax: The ELGDF Language*, Technical Report, Computer Science Department, Oregon State University, June 1988
- [Yon86] Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama, *Object-Oriented Concurrent Programming in ABCL/1*, OOPSLA 86 Conference Proceedings, pp. 258-268, 1986
- [Yok86] Yasuhiko Yokote, Mario Tokoro, *The Design and Implementation of ConcurrentSmalltalk*, OOPSLA 86 Conference Proceedings, pp. 331-340, 1986
- [Yon87] Akinori Yonezawa, Mario Tokoro, *Object Oriented Concurrent Programming*, The MIT Press, pp. 1-7, 1987
- [Agh87] Gul Agha, Carl Hewitt, *Actors: A conceptual Foundation for Concurrent Object-Oriented Programming*, Object-Oriented Concurrent Programming, The MIT Press, pp 49-74, 1987

- [Lie87] Henry Lieberman, *Concurrent Object-Oriented Programming in Act 1*, Research Directions in Object-Oriented Programming, The MIT Press, pp. 9-35, 1987