

OREGON STATE

UNIVERSITY

COMPUTER

SCIENCE

DEPARTMENT

Parallel Programming and Designing in Object Oriented Environment SS/1

Sungwoon Choi

Tom Sturtevant

Ted G. Lewis

Computer Science Department

Oregon State University

Corvallis, Oregon 97331-3902

90-60-9

# Parallel Programming and Designing in Object Oriented Environment SS/1

Sungwoon Choi, Tom Sturtevant, Ted G. Lewis

Computer Science Department

Oregon State University

Corvallis, Oregon 97331

## ABSTRACT

Parallel software development requires the flexibility to describe algorithms regardless of hardware specification, the ability to accomodate existing applications, and maintainability throughout the software life cycle. We propose the following model to address these issues. Our model incorporates aspects of the object-oriented and large grain data flow programming paradigms, and introduces a concept called a "Server". "Servers" are objects as well as self-contained processes which communicate with each other by sending messages. The server paradigm considers all components of a program as servers. This concept helps in designing flexible and dynamically reconfigurable software. The major goals of the server model are reusability, maintainability, and productivity. These are realized through encapsulation, instantiation, and inheritance features of the server model, as well as a graphical design environment with the capability of tracing and debugging the user's design based on the data flow information.

## 1. Introduction

Parallel processing is emerging as a promising way of computation as powerful new multiprocessor computers are becoming available at reduced cost. It also exaggerates the critical problems of productivity and maintainability of software because of the increasing architectural complexities. Several new paradigms for parallel programming have been introduced to cope with these problems. These include the process model based on communicating sequential processes [Hoa78], data flow [Ager82] and object-oriented

[Kay77] programming paradigms. In the communicating sequential process model, processes are executed sequentially and communicate with each other using communication channels. Dynamic interconnection topology is not supported in the process model, so the user must precisely specify the communication topology between processes as in OCCAM[Pou87]. This makes it difficult for the user to program using a large number of processors.

The data flow paradigm is based on dataflow machines which are programmable computers where the hardware is optimized for fine-grain data-driven parallel computation [Veen86]. This may fit fine grain parallel architecture in some applications containing no complicated control threads, but still shows a limitation in describing the nondeterministic nature of a program because of the fixed interconnection topology [DiN85].

In the object-oriented programming, a problem is modeled as a set of cooperating objects, where communication is achieved by exchanging messages among objects. In parallel programming, a problem is modeled as a set of cooperating processes. In that context, object oriented programming seems to fit naturally with parallel programming; objects correspond to processes and message passing corresponds to inter-process communication. In the usual object oriented programming paradigm like Smalltalk-80 [Gol83], it is not easy to detect implicit parallelism because the objects are created and deleted dynamically and context is encapsulated within the objects [Chu89]. There have been previous attempts to introduce external parallel programming constructs to the object oriented parallel programming paradigm e.g., Concurrent C++ and ConcurrentSmalltalk [Yok87]. These are awkward to use and hard to read, or unable to handle existing code.

The SS/1 (Server System/1) provides a framework for a parallel programming design

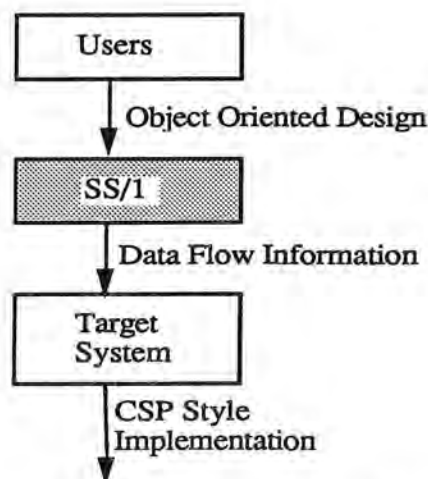


Fig 1.1 Design Flows in SS/1

environment based on object oriented programming which can be used with any existing programming language. It extracts data flow information from the user's application and schedules the application based on the communicating sequential process. (Fig 1.1) SS/1 considers the parallel programs as a collection of concurrently executable program modules, called "Servers". These interact with one another by sending messages and are used to define class hierarchy among objects, large-grain data flow architecture, and arbitrary source code modules.

In terms of the implementation, a server is similar to an Actor in the Actor model [Agha86]. In the Actor model, a process is realized as an Actor, and is dynamically created. When a process is created, it waits until it receives a message. Interaction between processes is in terms of message passing.

Although both models consider an object to be a self-contained process, they have these fundamental differences: 1) a server process is a transient process i.e., it has no state (data), 2) in the server model, the process synchronization scheme is based on data flow, while in the Actor model, processes are synchronized based on the message communication of the Actor. In other words, a server can be considered as a set of functions in the data flow programming paradigm. The message passing mechanism is described and analyzed using data flow information. This feature helps detect the implicit parallelism and validates external parallelism in user applications. It also makes SS/1 easy to use and flexible, because any other paradigms e.g., object oriented, large grain data flow or procedural programming paradigm, can be easily transformed into SS/1.

Language	Activation	Synchronization	Interconnection	Communication
CSP	parbegin/ parend	blocking send and receive	static	I/O commands through channels
DF	receiving data	data	static	data path
Actor	receiving a message	future message	dynamic	message passing and remote procedure call
SS/1	receiving a message	message	dynamic	message passing

Table 1.1 Comparison of SS/1 with other parallel programming languages

## 2. General Design Considerations

### 2.1 Design Considerations

#### 2.2.1 Design Layers

Object oriented programming is believed to be one of the best programming concepts to cope with computational complexities while providing features like maintainability, extensibility, and reusability [Mey88]. However it is not so much a coding technique as it is a code packaging technique, a way for code suppliers to encapsulate functionality for delivery to consumers [Cox87]. In this paper, the object oriented way of programming is introduced only at the large grain level. Because heavy message traffic in the fine grain level results in increased complexities and poor run time performance. The dynamic nature of message passing makes it almost impossible to automatically detect parallelism in the fine grain level.

The main idea in SS/1 is to decompose a program functionally based on the object oriented concept. At the fine grain level the programmer is free to use any special languages which fit their specialty, that is, APL can be used in vector computation and Prolog in AI applications. This customization makes it easy to build the automatic parallelization compiler with the least amount of effort and facilitates the reusability of existing code.

#### 2.1.2 Encapsulation and data flow information

Object oriented design facilitates the construction and use of reusable software components through support for data abstraction, generic operations, and inheritance. The benefits of data abstraction are achieved using encapsulation, which prevents an object from

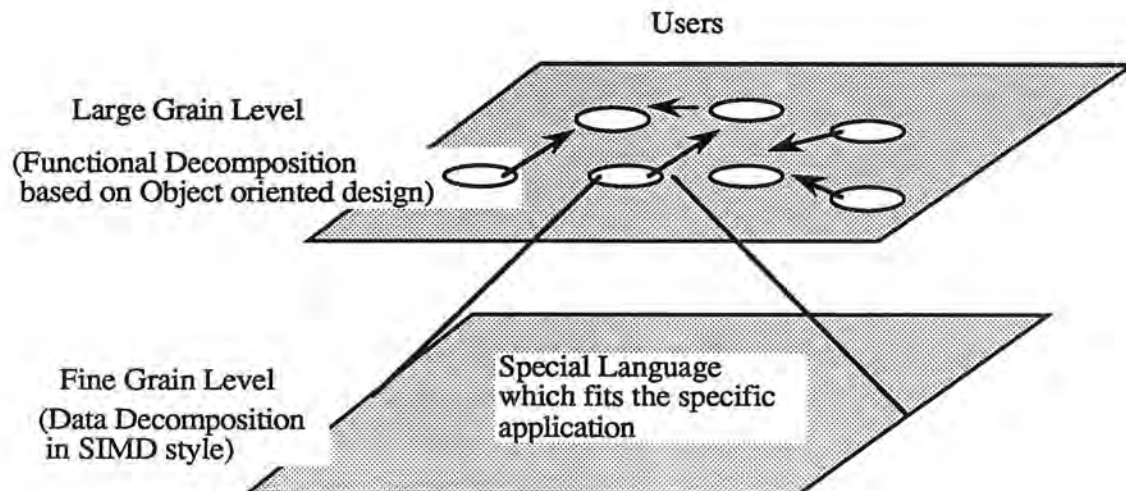


Fig 2.1 Design layer in SS/1



being manipulated except via its defined operations [Syn87]. The most important information in parallelizing an algorithm is the data flow information of the encapsulated instance variables. Because of this conflict between data encapsulation and data flow information, it is very difficult to directly map the object oriented design onto specific parallel hardware.

SS/1 sacrifices some advantages of instance variable encapsulation by moving them to an external interface. Therefore, the state of each server is stored in the external interface, where the system analyzes and extracts data flow information. This restriction may be harmful in the sense of encapsulation, but is very useful in parallel process synchronization, deadlock prevention and scheduling. It also gives more flexibility than the pure object oriented programming paradigm because any conventional programming paradigm as well as the object oriented programming paradigm can be mapped into SS/1.

### 2.1.3 Reconfigurability and extensibility

SS/1 is an open system. Modification of a system does not necessitate the reconfiguration or the redefinition of the entire system. Even when introducing a new entity, the system grows gracefully. Because each server, including its methods and event-handlers, is an independent process, they can be created and deleted dynamically. Section 3 shows a computer store simulation example program which uses a "Store" class hierarchy. Suppose that a new method "moneyOut" is created in the class "Environment". The only thing that must be updated is the method table in the class "Environment". This can be done during the execution of a system by receiving the message from the new method "moneyOut". The system provides a dynamic interconnection topology.

### 2.1.4 Portability

SS/1's design environment consists of the design editor and a run time environment. A graphical design editor produces glue code written in a meta language called "SML" (Server Manipulation Language) [APPENDIX I] from the user's design. The run time environment will execute the user's design in a target machine by interpreting the SML language. SML has only a minimum set of constructs and assumes an abstract machine. The run time environment of a system can be developed in a target machine based on SML by defining the process communication and the process creation mechanism. Implementation details of a method is handled by the existing compilers and system. The method in SS/1 is treated as a independent light weight process of the target system. SML gives SS/1 the power of portability among specific hardware configurations.

### 2.1.5 Reusability

Reusability of the existing code is one of the most important issues in computer science. It has already been mentioned that the methods in SS/1 are independent programs. That is, any existing program can be reused in SS/1 with no changes. Especially in reverse engineering, SS/1's methods are mapped into the functions or procedures in conventional programming languages.

## 2.2 Basic Constructs

### 2.2.1 Servers

SS/1 introduces a new concept called "Servers" which are self-contained parallel agents communicating with each other by sending messages. A server is an object plus one or more processes. Once a server is instantiated from a class, the server becomes an active computational agent which carries out its actions in response to incoming messages. A server consists of an event-handler process and one or more method processes (see Fig.2.2). The event-handler process reads messages from its message queue and looks for appropriate methods in its method table. If the requested method exists, the event-handler process spawns another process for executing the method. If no method is found, a search up the class hierarchy is made to find the appropriate method to be inherited from a superclass.

A method process can be duplicated as many times as the user wants. In contrast to other object oriented programming paradigms [Gol83], method processes may not have persistent state, because they may be executed only once and then lose their state information.

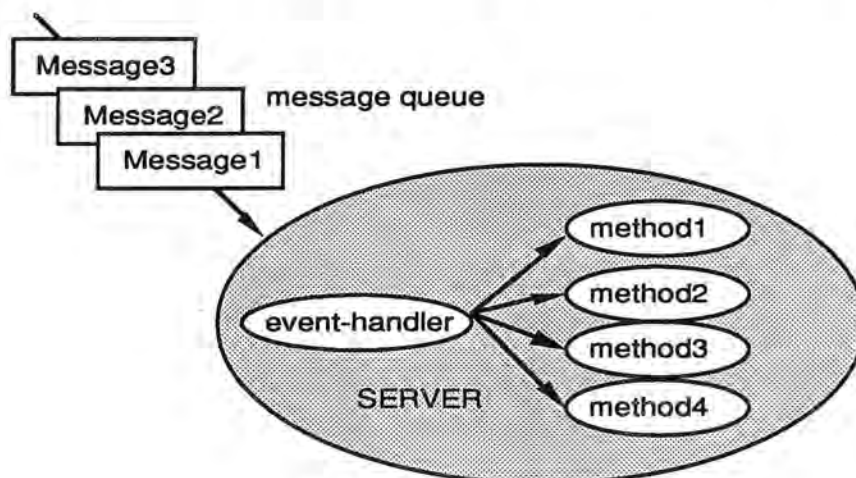


Fig 2.2 Servers and messages

All transient state information is stored in a global communication.

Each server maintains a scheduler which schedules its method processes by determining processor and communication overhead. Incoming messages are queued in the message queue and scheduled dynamically according to the current system overhead. This kind of distributed scheduling finds local and current optimal scheduling. Optimal static scheduler may be estimated by the other tools[Hes89].

### 2.2.2 Message and synchronization

A message handler is a self-contained process which is responsible for the delivery of messages and the synchronization of the system (see Fig 2.3). Once a message handler process is activated by SS/1, it sends its message content to the target server. The messages are queued in the message queue of the target server and the event-handler of the target server reads that message and creates and activates the appropriate methods. The message handler processes wait for termination signals from the methods. If the termination signal arrives, the message handler process terminates itself.

Basically the process synchronization scheme in SS/1 is synchronous, whenever a message process is created, SS/1 waits for its termination. However, asynchronous processes can also be created by parallelizing message handler process activation.

In a message passing machine, the message handler process can be a communication manager which controls the routing of its messages and data depending on its interconnection topology and communication overhead. Even in a shared memory machine, message handler processes have information about shared memory and can guarantee mutual exclusion and prevention of deadlock.

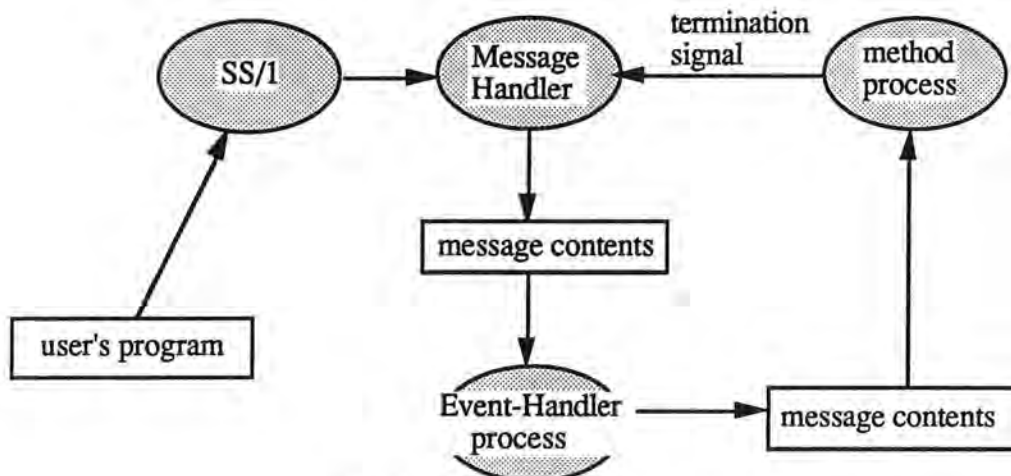


Fig 2.3 Message process



### 3. System Architecture

#### 3.1 Overview

SS/1 consists of a design editor and a run time environment (Fig 3.1). The design editor provides the object oriented data flow graphical language and generates the SML code from the user's design. Then SS/1 translates the SML source code and executes the user code on the target machine. Although SS/1 is basically standalone, it can be used in combination with other tools. In particular, the communication specification can be converted to TaskGrapher[Hes89] format so that it may be analysed by TaskGrapher which generates a static schedule and mapping of tasks onto processors.

For our experiment, the design environment is built on an Apple Macintosh and the target machine is a Sequent Balance 21000 system.

#### 3.2 Design editor

The SS/1 Design Editor (Fig 3.2) is an integrated Macintosh application used to describe the Class Definition, as well as the Communication Specification.

##### 3.2.1 Design Editor features [APPENDIX II]

- Facilitates Top-Down/Hierarchical design.
- Built-in word processor for editing method definitions.
- Built-in translator generates glue-code for target machine, as well as TaskGrapher format files.

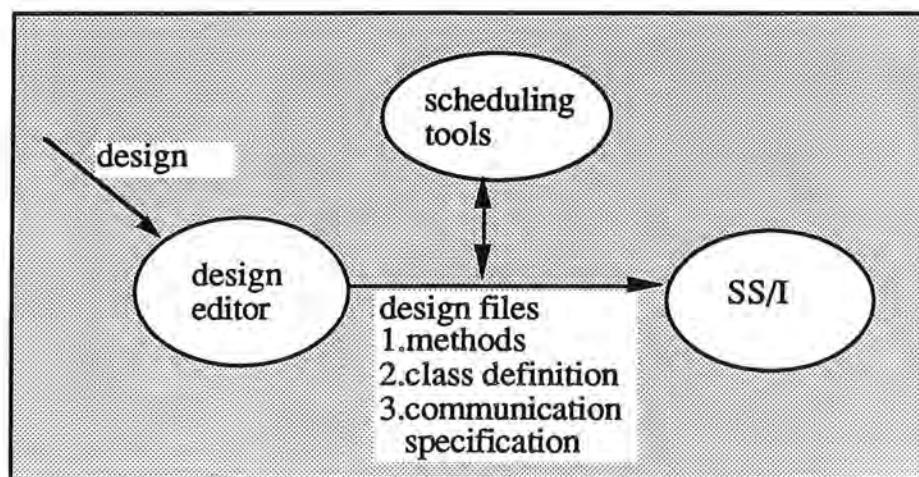


Fig. 3.1 data flow diagram for the server system.

### 3.2.2 Design Constructs

The design environment in SS/1 consists of two parts, the Class definition, and the Communication specification. In the class definition, a class hierarchy is defined using a tree structure. The left window of Fig 3.2 shows the class hierarchy where terminal nodes are methods and internal nodes are classes. A class node has only a name and class hierarchy information while a method node contains code which defines the behavior of a method process.

In the communication specification, the user can program in terms of message passing, inheritance, and the data flow. The right window of Fig 3.2 shows the communication specification of the matrix multiplication problem. A message consists of target object name, method name, and its arguments. Fig. 3.3. shows the description of the communication specification context palette. There are four kinds of messages in SS/1, e.g., "simple", "compound", "serial replicated", and "parallel replicated". The simple message indicates the point where the actual message is sent to the target object, whereas the compound message is a high level design construct which represents a set of messages linked together. The serial replicated and the parallel replicated messages are the messages generated repeatedly in serial or parallel. Using the concept of messages, mutual exclusion and prevention of deadlock can be guaranteed because message creation is based on data dependency information.

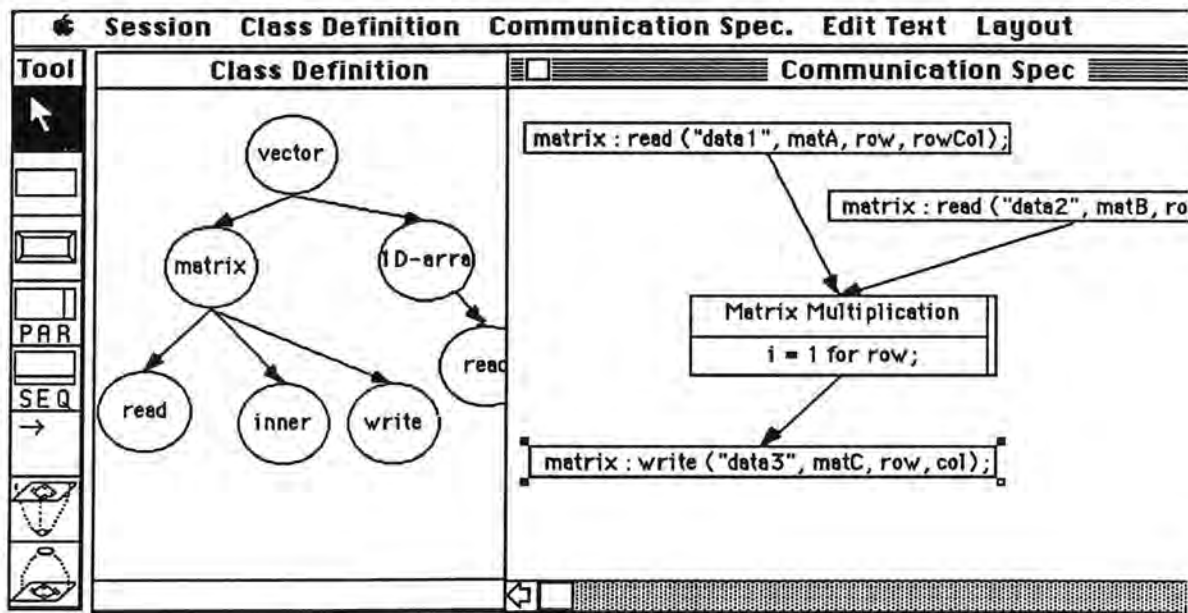


Figure 3.2 The SS/1 Design Editor



Fig 3.3. Communication Specification Context Palette

### 3.2.3 The Design Cycle

The user first draws the class definition hierarchy as shown in fig 3.2 (an existing class hierarchy file may be used). A method definition is then specified for each method in the class hierarchy. The definition can be a procedure in any compilable language, it can also be a UNIX system call or a Macintosh ToolBox routine.

The next step is to describe the communication specification (also shown in fig 3.2). Icons representing simple messages, compound messages, and replicated messages are connected with arcs to indicate data dependency relation. The built-in translator then generates SML code describing the class definition and the communication specification. This code along with the method definition files is then moved to the target machine.

## 3.3 Run time environment

The SS/1 run time environment consists of three components, Translator, Event-Handler and Methods.

### (1) Translator

The translator parses SML code and activates the appropriate message handler processes as specified in SML code. There is a one-to-one correspondence between the program constructs and the message handler processes. Each message handler process delivers its own portion of the program messages to the target server.

## (2) Event-handler

The event-handler can be viewed as a method manager. When it is activated, it sets up the method table which has the physical process id associated with the logical method name in the Server. It reads a message from its message queue and activates the appropriate method process. It is responsible for the scheduling of the method processes. It schedules the method processes dynamically according to the current system overhead, i.e., allocate the method process to the processor with the least overhead. Once the method process is activated, the event-handler loses all information about the method process. Method inheritance and overriding are also handled by the Event-handler.

## (3) Methods

A method is a procedure written by the user and linked with the communication code supported by the system. Each method is considered as an independent process. Once the method process finishes its job, it sends a "termination" message to its message process and is terminated. When it is terminated, it loses all its state information. Only the communication variables which are defined in the external interface of the method keep the result of the execution. Each method can be duplicated to invoke multiple instances of the method.

# 4. Design Example : Computer Store Simulation

## 4.1 Class Definition

To design a computer store simulation program, the "Store" can be defined as a super class of all the other classes and is decomposed into the subclasses, "Ware", "People", and "Environment". Each subclass can be subclassed again. For example, the class "people" have three subclasses, "Service", "Sales", and "Customer". The class "Sales" has two methods "demo" and "sell". Fig 4.1 shows a possible class definition using the graphical editor of SS/1.

## 4.2 Communication Specification

Fig 4.2 shows the communication specification of top level computer store design which can be interpreted as follows,

- Open the store

- sales persons, service persons, and the customers do their role in parallel
- Close the store

The "Sales Loop", "Service Loop" and "Customer Loop" messages is processed in parallel after the "Environment : open();" message is processed. After the all message is processed, the "Environment : close();" message is processed. The arc represents the flows of data between messages.

The parallel replicated compound message "sales loop" will be replicated for MAX\_SALES times in parallel and its lower level will describe the specific behavior of a single person. The "service loop" and "customer loop" can be interpreted in the same way.

Fig 4.3 shows the behavior of a individual sales person and can be interpreted as

- depending upon which event happened (conditional branch)
- send the messages "sell" or "demo" to the object "Sales"

Fig 4.3 shows the non deterministic flow of data between the methods "getEvent" and "sell" and the methods "getEvent" and "demo". The non-deterministic arc is shown as a dotted arc and means that the next message is sent depending on the data from the previous message.

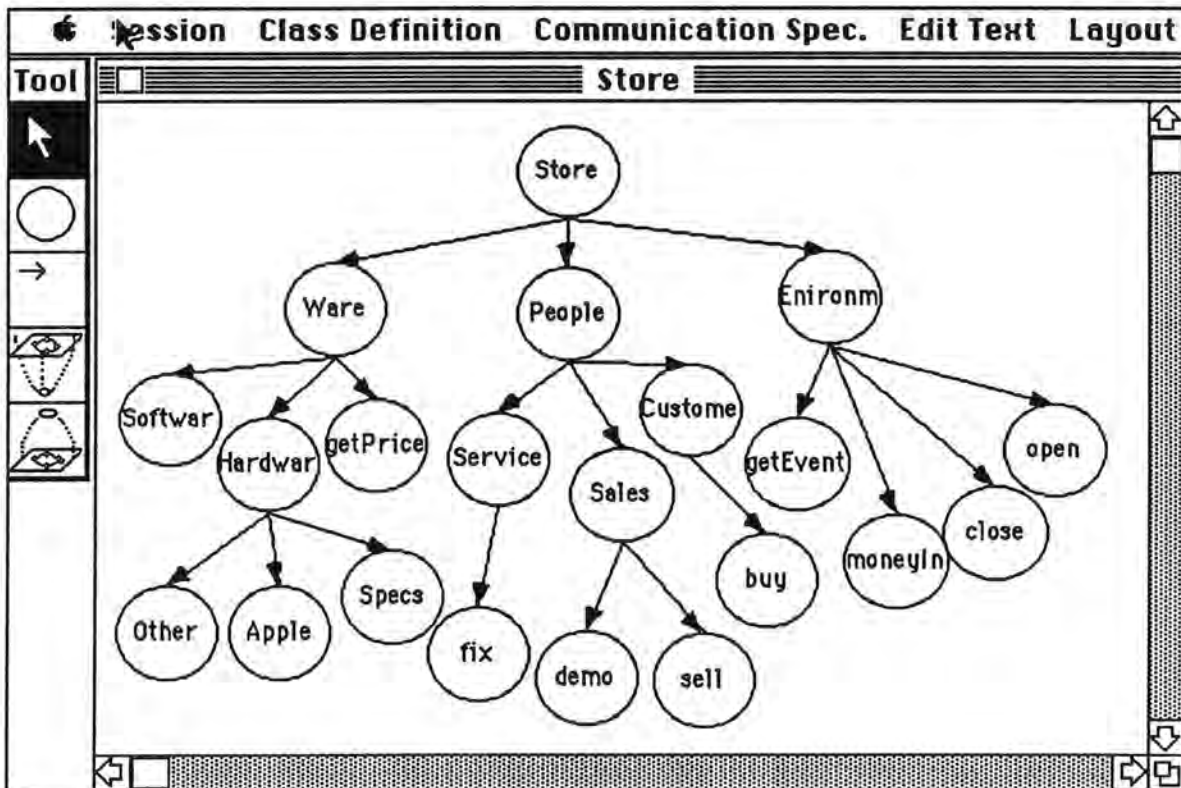


Fig 4.1 Class Definition



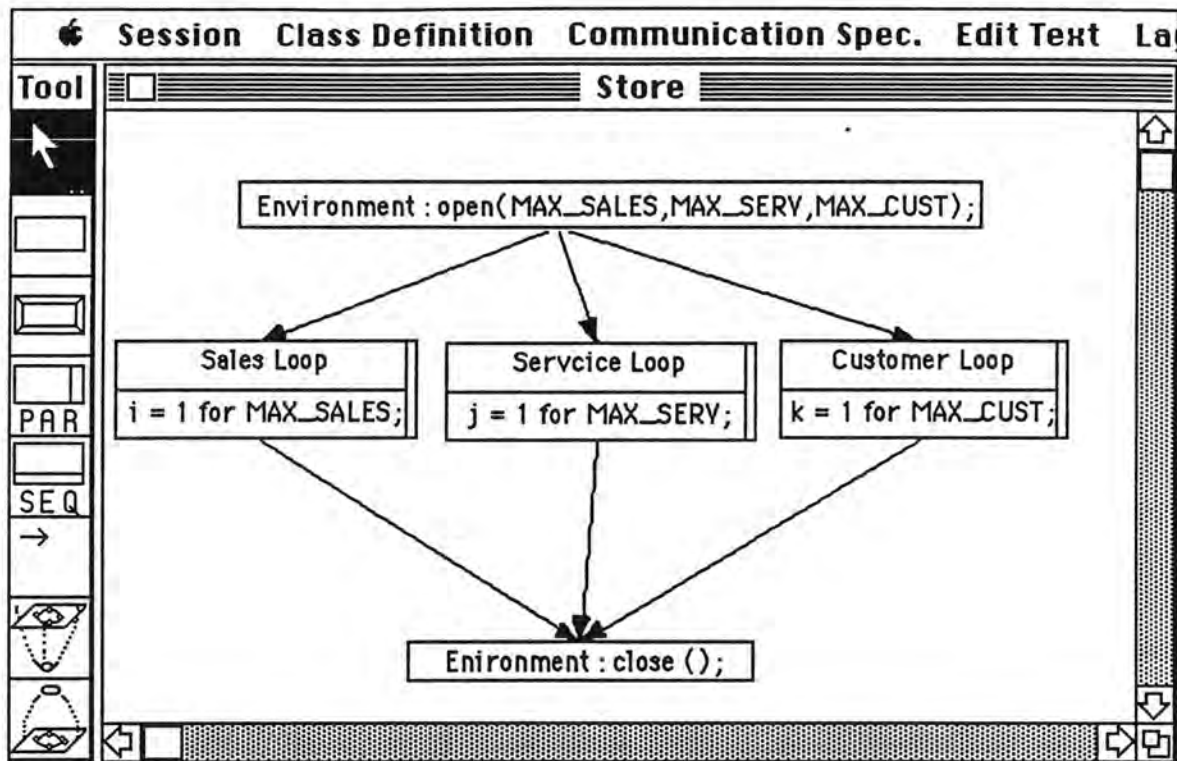


Fig 4.2 Communication Specification : Level 1

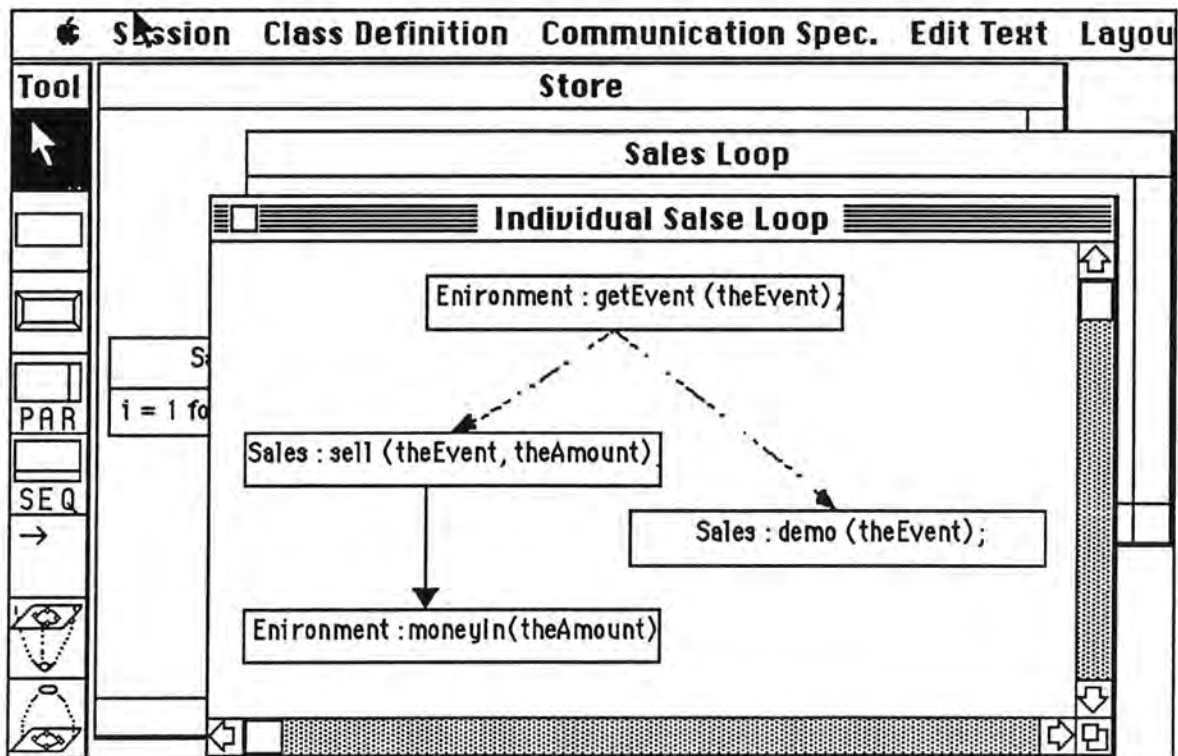


Fig 4.3 Communication Specification : Level 2

Once the user finishes a design, the class definition and communication specification data are automatically translated into SML. The SML description also consists of the class hierarchy definition and the message communication specifying the control and data flow of the messages to be sent. The SML source code generated by the editor is shown in APPENDIX III.

#### 4.3 Message and Control Flow : Computer Store Simulation

To execute the SML code in the SS/1 environment, the methods must be compiled and linked with a communication handler which handles the message passing mechanism in the methods. Once all of the methods are ready as in the class definition part, the system can be executed. The parser reads the class definition and initializes the methods and classes. The system code "event" will be woken up as the class processes and all methods will be run as light-weight independent processes. After the initialization is finished, the parser executes the communication specification in the SML code.

In the next section, the execution details of the system will be shown using a "PAR-SEQ tree" which is the mapping of SML implementation. This looks like an "and-or tree", but it represents the process creation order. The nodes represent processes, and the edges indicate control flow (Fig. 4.4.(a)). A child node is always created by its parent node. Once the parent node wakes up its child process, it waits for the termination of the process.

There are two ways to create children nodes. The "SEQ" subtree as in Fig 4.4 (b) with arcs across the edges shows the sequential process creation order, from left to right. In this case, the parent process creates its child processes and waits for them to terminate one by one. This corresponds the SML construct, "SEQ" which represents the serial execution of its sub structures.

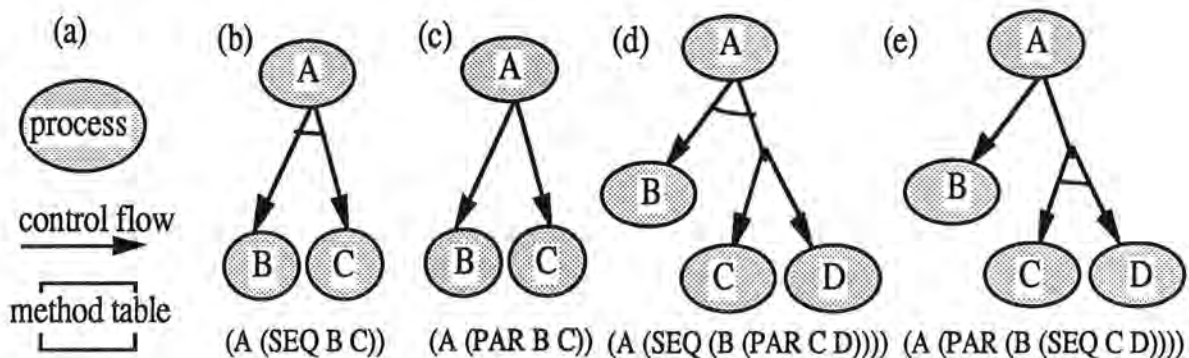


Fig 4.4 PAR-SEQ tree

The "PAR" subtree as in Fig 4.4 (b) which looks like normal tree means the parallel creation of the child process nodes. The parent process wakes up all its children processes at the same time and waits for the terminations of all the created processes. This corresponds the SML construct, "PAR" which represents the parallel execution of its sub structures. One special form of the "PAR\_SEQ" tree is the edge splitting which discriminates the message creation order among the subtrees. Fig 4.4 (d) is the PAR-SEQ tree representation of (SEQ A (PAR B C)) and (e) comes from (PAR A (SEQ B C)).

#### 4.2.1 Class Definition Phase

A part of the computer store class hierarchy can be represented as follows using SML.

(Store (People (Sales (demo sell))))

The super class "Store" has subclass "People". The class "People" has "Sales". And the class "Sales" has two methods "demo" and "sell". According to the definition, SS/1 will set up the process class hierarchy on the following order (Fig 4.5).

- SS/1 wakes up the class processes as defined in the class definition.
- Each class process sets up a method-table using the information from its methods.
- The system sets up the communication buffer and unique process IDs.

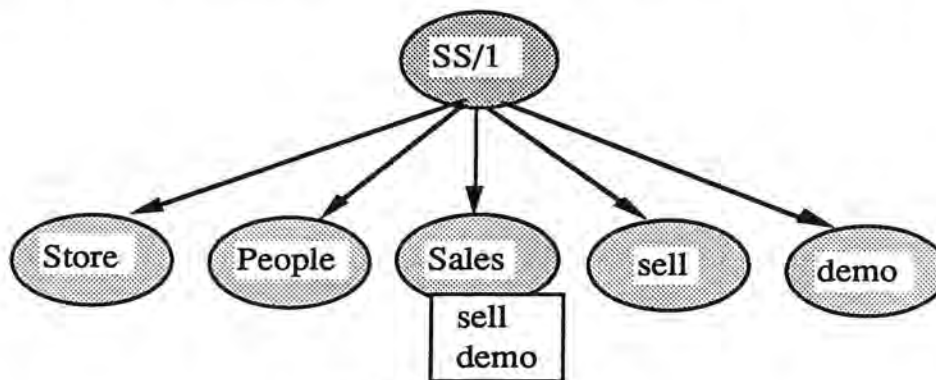


Fig 4.5 Initialization Phase

#### 4.2.2 Communication Specification Phase

The followings are the communication specification of the salesman's behavior.

- 1) (PAR i = 1 for MAX\_SALES
- 2)     (SEQ a = 1 for forever
- 3)         (SEQ
- 4)             (Environment, (getEvent,theEvent))
- 5)             (PAR

```

6)                (SEQ
7)                (Sales, (sell,theEvent, theAmount))
8)                (Environment, (moneyIn,theAmount)))
9)                (Sales, (demo,theEvent))
) ) ) )

```

SS/1 interprets the above SML code in the following way.

- SS/1 reads SML source code, "1) PAR i=1 for MAX\_SALES" and forks the message process MAX\_SALES times as in level (a) of fig4.6.
- Each message process executes from "2) SEQ a= 1 for forever". (level b of fig4.6)
- The message process sends the "4) getEvent, theEvent" message to the object process "Environment". (level b, c of fig4.6)
- The message process forks two other message processes. (5-9)(level b, c of fig4.6)
- One message process executes from "6) SEQ" and sends the message "7) sell, theEvent, theAmount" to the object process "Sales". Once the message is processed, it sends "8) moneyIn, theAmount" message to the object process "Environment". The message process is terminated.(level c, d of fig4.6)
- The other message process sends the "9) demo, theEvent" message to the object process "Sales". (c, d of fig4.6)

LEVEL

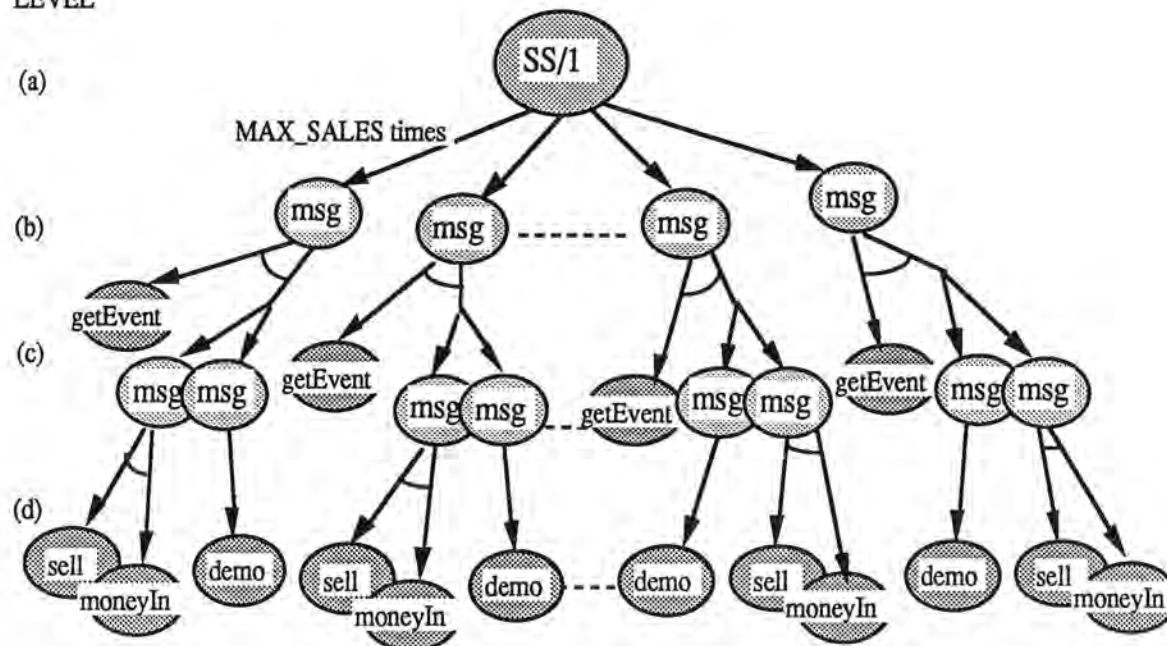


Fig 4.6 Message flow and process creation

## 5. Conclusion

The computational model presented in this paper is based on three programming paradigms : Communicating sequential processes, Data flow, and Object oriented programming. The notion of "Server" provides a sound foundation for massively concurrent object-oriented programs, and transparent object-oriented design environments.

SS/1 is designed for the general computation and may have more overhead than the customized language for the specific hardware like Parallel-C[Seq88]. However SS/1 shows good run time behavior compared with Parallel-C even in very fine grain heavy computational application. APPENDIX IV shows the performance comparison between SS/1 and Parallel-C in Sequent Balance.

However, the concept of server is in its infancy, and there remain many issues to be discussed.

### 5.1 Scheduling

SS/1 uses dynamic scheduling. It schedules the processes to processors in first-come-first-serve fashion, considering the overhead of communication and processors. Unfortunately this does not guarantee optimal scheduling. A more intelligent scheduling scheme must be developed, e.g., static scheduling based on dynamic behavior of processes and processors.

### 5.2 Data parallel [Dan86]

SS/1 is well suited for large grain or medium grain size problem decomposition. It is not designed to deal with the fine grain parallelism. Many useful techniques are already introduced to automatically detect parallelism from serial program in fine grain level, e.g., loop spreading [Wu88][Hes89]. Among them, data parallel language is one of the best concept to cope with fine grain parallelism. They can be combined with SS/1 in the fine grain level with automatic data partitioning and parallelization technique.

### 5.3 SS/1 as a multi programming environment

SS/1 can support the procedural and functional programming environments as well as the object oriented programming environment, because SS/1 provides a means of programming, not a way of coding. SS/1 can give more flexibility in the field of reverse engineering where existing procedural languages are mapped to parallel environments. (fig



5.1)

#### 5.4 SS/1 in various kinds of hardware architecture

The prototype system of SS/1 now runs on the Sequent Balance which is a typical shared-memory machine. It should be ported to various parallel architectures.

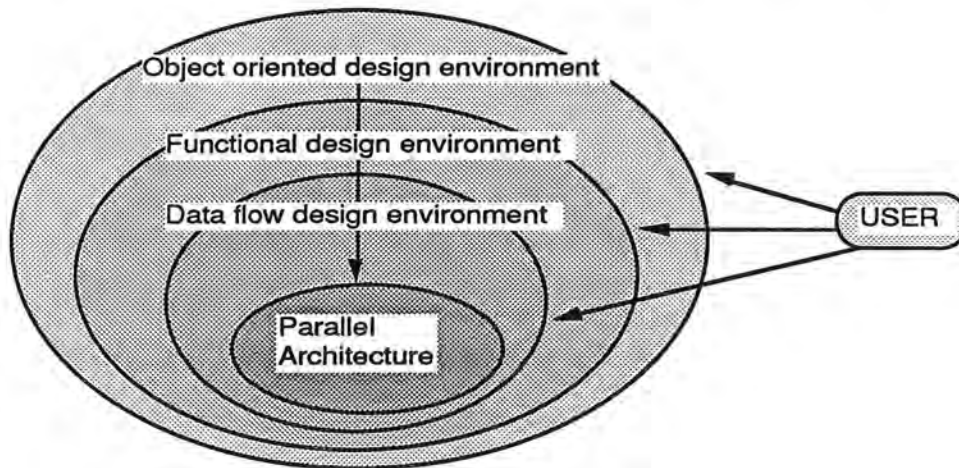


Fig 5.1 SS/1 as a multi language paradigm

## Appendix

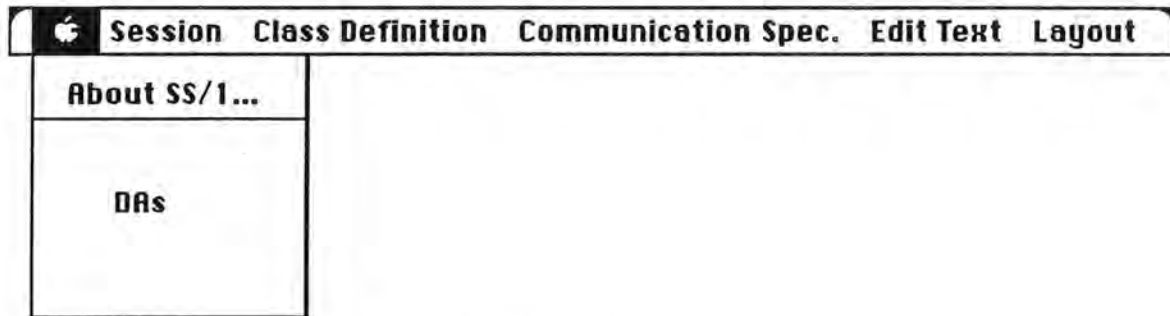
## I. SML Specification

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

## II. Design Editor User Interface

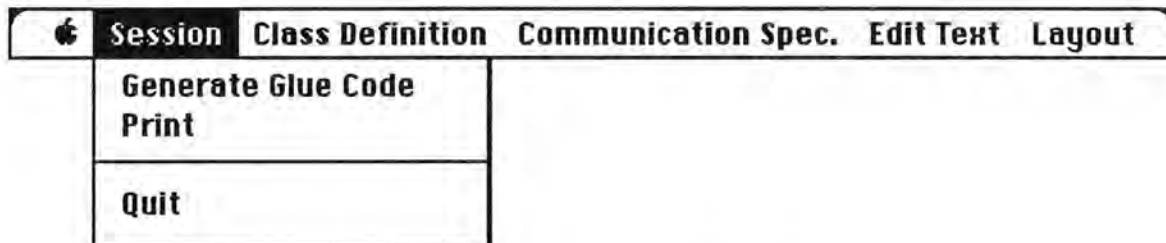
### 1. Design Editor Menus

#### 1.1 Apple/About Menu



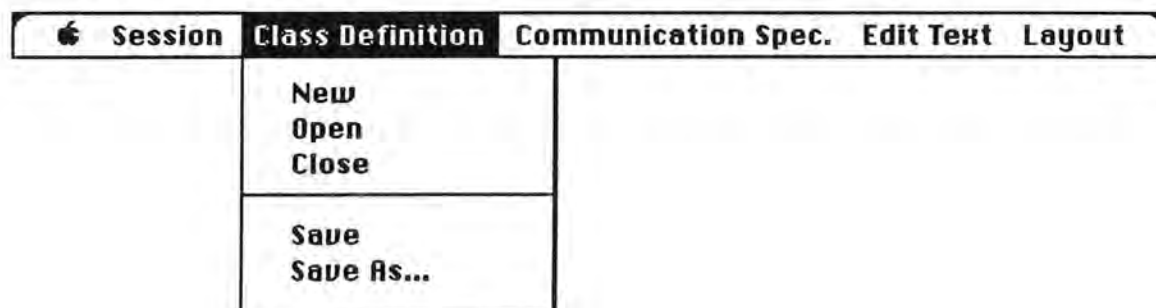
- About SS/1 Display the initial SS/1 dialog.
- Desk Accessories System desk accessories are fully supported.

#### 1.2 Session Menu



- Generate Glue Code Generate SML description of the design.
- Print Print active window (Text or Graphics).
- Quit End design session.

#### 1.3 Class Definition Menu




- New Open a new Class Definition window.
- Open Open an existing Class Definition file.
- Close Close the current Class Definition window.
- Save Save the Class Definition information.
- Save As Save the Class Definition in the file specified.

#### 1.4 Communication Specification Menu

 Session	Class Definition	Communication Spec.	Edit Text	Layout
		New Open Close		
		Save Save As...		
		Export Task Graph As...		

- New Open a new Communication Specification window.
- Open Open an existing Communication Specification file.
- Close Close the current Communication Specification window.
- Save Save the Communication Specification information.
- Save As Save the Communication Specification in the file specified.
- Export Task Graph As Generate a Task Graph description of the Communication Specification.

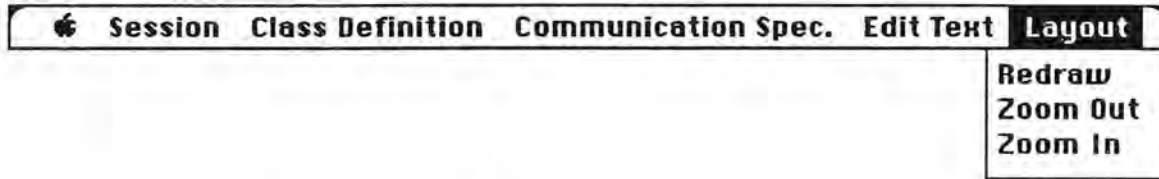
#### 1.5 Edit Text Menu

 Session	Class Definition	Communication Spec.	Edit Text	Layout
			Cut Copy Paste	
			Save Save As...	

- Cut Delete currently selected text. Place this text in the paste buffer.
- Copy Place the currently selected text in the paste buffer.
- Paste Insert the contents of the paste buffer at the current position.

- Save Write the text to the Method Definition file.
- Save As Write the text to the specified Method Definition file.

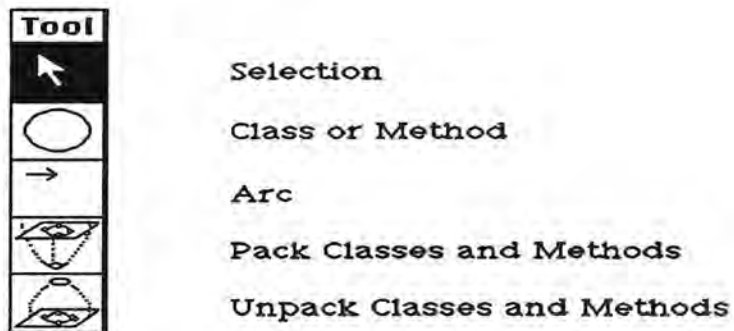
## 1.6 Layout Menu



- Redraw Repaint the current window.
- Zoom Out Decrease the drawing scale of the current window.
- Zoom In Increase the drawing scale of the current window.

## 2. Design Editor Tool Palettes

### 2.1 Class Definition Context Palette



### 2.2 Communication Specification Context Palette





## III. SML source code of Computer Store Simulation Example.

```

/* Class Definition */
(
  Store (
    Ware (
      Software
      Hardware ( Other Apple Specs)
      getPrice
    )
    People (
      Service ( fix )
      Sales (demo sell)
      Customer (buy)
    )
    Environment (getEvent moneyIn open close )
  )
)
/* Communication Specification */
(SEQ
  (Environment, open)
  (PAR
    (PAR i = 1 for MAX_SALES
      (SEQ a = 1 for forever
        (SEQ
          (Environment, (getEvent,theEvent))
          (PAR
            (SEQ
              (Sales, (sell,theEvent, theAmount))
              (Environment, (moneyIn,theAmount))
            )
            (Sales, (demo,theEvent))
          )
        )
      )
    )
    (PAR j = 1 for MAX_SERV
      (SEQ b = 1 for forever
        (SEQ
          (Environment, (getEvent,theEvent))
          (Service, (fix,theEvent, theAmount))
          (Environment, (moneyIn,theAmount))
        )
      )
    )
    (PAR k = 1 for MAX_CUST
      (SEQ c = 1 for forever
        (SEQ
          (Environment, (getEvent,theEvent))
          (Customer, (buy,theEvent, theAmount))
        )
      )
    )
  )
  (Environment, close)
)

```

IV. Comparison of performance between SS/1 and parallel C on the Sequent Balance  
 - Matrix Multiplication Problem (Timing unit : second)

(1) Number of processors used = 10

<u>matrix size</u>	<u>SS/1</u>	<u>Parallel C</u>
50	5.88	4.10
100	17.35	15.25
200	79.49	77.86
300	222.37	218.88

(2) Number of processors used = 20

<u>matrix size</u>	<u>SS/1</u>	<u>Parallel C</u>
50	6.70	4.52
100	15.33	13.46
200	60.74	59.12
300	158.82	154.32

(3) Number of processors used = 27

<u>matrix size n</u>	<u>SS/1</u>	<u>Parallel C</u>
50	6.78	4.84
100	15.45	14.60
200	58.79	57.45
300	145.88	145.22

## References

- [Ager87] T. Agerwala, Arvind, Data flow systems, Computer 15:2, Feb 1982.
- [Agha86] G. A. Agha, ACTORS : A Model of Concurrent Computation in Distributed Systems, MIT Press. 1986.
- [Chu89] Jean Chung, Implicit Parallelism in Object Oriented programming , Technical Report, Computer Department, Oregon State University, November 1990.
- [Cox87] Brad J. Cox, Object Oriented Programming : An Evolutionary Approach, Addison-Wesly, 1987
- [Dan86] W. Daniel Hillis, Guy L. Steele, Jr., Data parallel algorithms, Communications of the ACM 29,12, December 1986.
- [DiN85] David C. DiNucci, Robert G. Babb II, Design and Implementation of Parallel Programs with LGDF2, Characteristics of parallel algorithms, Cambridge, MA: MIT Press,1985
- [Hes89] Hesham El-Rewini, Task Partitioning and Scheduling on Arbitrary Parallel Processing, Ph.D dissertation, Oregon State University, November 1989.
- [Gol83] Goldberg, A., D.Robson, Smalltalk-80: The Language and its Implementation , Addison-Wesley, 1983
- [Hew73] Hewitt, C., et al., A unversal, Modular Actor Formalism for Artificial Intelligence , Proc. of IJCAI, 1973.
- [Hew77a] C. Hewitt and R. Atkinson, Synchronization in actor systems, Proceedings of Conference on Principles of Programming Languages, pages 167-280, January 1977.
- [Hew77b] C. Hewitt and H. Baker, Laws of communicating parallel processes, 1977 IFIP Congress Proceddings, pages 987-992, IFIP, August 1977.
- [Hoa78] C. A. R. Hoare, Communicating sequential processes, CACM 21:8, 666-677, August 1978.
- [Kay78] Alan Kay, Microelectronics and the Personal Computer, Scientific American, Vol 237: 230 244, September 1977.
- [Mey88] Bertrand Meyer, Object-Oriented Software Construction,Prentice Hall, 1988.
- [Pou87] Pountain, D., A Tutorial Introduction to OCCAM Programming, INMOS Ltd., March 1987.
- [Seq87] Sequent Computer Systems, Guide to Parallel Programming., Sequent Computer Systems, Inc. 1987.
- [Syn87] Alan Snyder, Inheritance and the Development of Encapsulated Software Systems, Research Directions in Object-Oriented Programming, The MIT Press,

1987.

- [Wu87] Youfeng Wu, Parallel Simplex Algorithms and Loop Spreading, Ph.D dissertation, Oregon State University, 1988
- [Yok87] Y. Yokote, M. Toloro, Concurrent Programming in ConcurrentSmalltalk, in Object-Oriented Concurrent Programming, The MIT Press, 1987.
- [Yon87] A. Yonezawa and M. Tokoro, Object-Oriented Concurrent Programming , pages 1-7, The MIT Press, 1987.