Chapter 7

A Process Model and its Hierarchical Refinements for Slicing of Concurrent Programs.

Static slicing techniques use static analysis to derive slices, i.e. the source code of the program is analyzed and the slices are computed for all possible input values. No assumptions are made about the input values. Therefore, conservative assumptions have to be made during slice computations, which may lead to relatively larger slices. However, as we have already discussed in Chapter 1, static slicing is helpful in several applications such as software maintenance, reuse, comprehension, integration and differencing, etc. Static slicing methods can also be used to detect dead code, i.e. statements which are never executed. Often such dead code get created because of the presence of a bug. Static slicing techniques can also be used to detect uninitialized variables which are used in expressions, another source of errors in a program.

As already stated in Chapter 3. Many researchers have contributed to enrich the techniques of static slicing of sequential programs. However, research efforts concerning development of static slicing techniques for concurrent programs are scarcely reported in the literature. In this chapter, we present a framework for static slicing of concurrent programs. In our static slicing framework, we consider concurrent programs in a Unix process environment. As a part of our methodology, we introduce the notion of a Concurrent Program...
Dependence Graph (CPDG). A CPDG represents various aspects of concurrent programs in a hierarchical fashion. This hierarchical representation lets us compute static slices of programs at different levels of abstraction. CPDGs can be used to represent concurrent programs written using Unix primitives. A CPDG is constructed in three levels. Once the CPDG of a multiprocess program is constructed, it becomes easy to compute its slice from the CPDG representation.

Based on our methodology, we have implemented a static slicing tool supporting an option to view slices of programs at different levels of details. Experience with our implementation shows that this approach helps the user get a better understanding of the behaviour of concurrent programs.

Our work focuses mainly on handling various aspects of concurrent programs like process creation, shared memory, message passing, etc. Therefore, we have not discussed handling of programming constructs such as arrays, pointers, and procedures in a concurrent programming framework. Arrays can be addressed either through a conservative approach as in [101] or by using the approach proposed by Ottenstein and Ottenstion in [87]. Handling of pointers in traditional programs has been discussed by Horwitz et al and Agarwal et al [135, 48]. These techniques can be easily integrated into our framework. Interesting reports on handling arrays and pointers have also been discussed by many other authors [41, 25, 147, 107, 70, 28, 73].

7.1 Preliminaries

In our subsequent discussions, we will use primitive constructs for process creation, interprocess communication and synchronization based on
those available in the Unix environment [63]. The main motivation behind our choice of Unix-like primitives is that the syntax and semantics of these primitive constructs are intuitive, well-understood, easily extensible to other parallel programming models and also can be easily tested.

The language constructs that we consider for message passing are msgsend and msgrecv. The syntax and semantics of these two constructs are as follows:

- **msgsend (msgqueue, msg):** When a msgsend statement is executed, the message msg is stored in the message queue msgqueue. The msgsend statement is nonblocking, i.e. the sending process continues its execution after depositing the message in the message queue.

- **Msgrecv(msgqueue, msg):** When a msgrecv statement is executed, the variable msg is assigned the value of the corresponding message from the message queue msgqueue. The msgrecv statement is blocking, i.e. if the msgqueue is found to be empty, the receiving process waits for the corresponding sending process for depositing the message.

We have considered nonblocking send and blocking receive semantics of interprocess communication because these have traditionally been used for concurrent programming applications. In this model, non assumptions are made regarding the order in which messages arrive in a message queue from the msgsend statements belonging to different processes except that messages sent by one process to a message queue are stored in the same order in which they were sent by process i.e., the message queue preserves the order of messages sent from any single process. A process executing a msgrecv
(msgqueue, msg) statement removes the first available message from the msgqueue.

A fork() call creates a new process called child which is an exact copy of the parent. It returns a nonzero value (process ID of the child process) to the parent process and zero to the child process [106]. Both the child and the parent have separate copies of all variables. However, shared data segments acquired by using the shmget () and shmat() function calls are shared by the concerned processes. Parent and child processes execute concurrently. A wait() call can be used by the parent process to wait for the termination of the child process. In this case, the parent process would not proceed until the child terminates.

Semaphores are synchronization primitives which can be used to control access to shared variables. In the Unix environment, semaphores are realized through the segment() call. The value of a semaphore can be set by semctl() call. The increment and decrement operations on semaphores are carried out by the semop() call [63]. However, for simplicity of notation, we shall use P(sem) and V(sem) as the semaphore decrement and increment operations respectively.

### 7.2 Representation of Concurrent Programs

This section introduces a method to graphically represent concurrent programs in Unix process environment. This representation is later used to compute static slices. Our method can handle only static creation of
processes. We construct the graph representation of a concurrent program through three hierarchical levels: process graph, concurrency graph, and CPDG. Construction of these three hierarchical levels is described next.

### 7.2.1 Process Graph

A process graph captures the basic process structure of a concurrent program. We represent the process creation, termination, and joining of processes in a graph we call a process graph.

**Definition 4.1 (Process Graph).** A process graph $G_p = (N_p, E_p, f_p)$, is a directed graph where $N_p$ is a set of nodes referred as process nodes. $E_p \subseteq N_p \times N_p$ is a set of edges. $f_p$ is a function assigning statements to process nodes. The edges of a process graph can be of two types: fork edges and join edges.

Informally, a process node consists of a sequence of statements of a concurrent program which would be executed by a process. A process node and a process are not the same. Let $S_q$ represents a statement sequence which is to be executed by a process. $S_q$ may be split into different subsequences e.g., $S_{q1}$, $S_{q2}$, ..., $S_{qk}$, where each of these subsequences represents a process node. In such a case, the last statement of each subsequences except the last i.e. $S_{qk}$ is a fork call. Three process creation programming examples are shown in figure 7.1. For each of the cases shown in figure 7.1, there are three process nodes: P1, P2 and P3. The statement sequence represented by each of these process nodes are provided within parenthesis (using integer labels) along with each node. For example $P_x(m,n)$ would mean that process $P_x$ contains statements numbered $m$ and $n$. Case (a):
SLICING OF CONCURRENT PROGRAMS

P1(1,2), P2(3,5) and P3(4,5). Case (b): P1(1,2), P2(3,4) and P3(4). Case (c): P1(1,2), and P2(3) and P3(3).

In figure 7.1(a), P1 and P3 together represent the parent process whereas P2 represents the child process. That is, the statement sequence (1,2,4,5) to be executed by the parent process is split into two subsequences, (1,2) and (4,5) each representing a process node. The last statement of process node P1, which is 2, is a fork call. The statement sequences to be executed by a process is examined from the beginning for any fork call. If any fork call is found, then the subsequence from the beginning to this fork call represents a process node. This process is continued until all the statements in the sequence are exhausted. This leads to a process node which represents the last subsequences which does not contain any fork call. Of course, this construction cannot handle fork calls within loops and conditionals. It is important to note that each statement belongs to one or more process node.

1. S1
2. if(fork()==0) 2. if(fork()==0) 2. Fork()
   else
4. S3 4. S3
5. S4

(a) (b) (c)

Figure 7.1: Three Typical Statement Sequences for Creating a Process.
A process node is created when there is a fork call in the program. A fork edge from a node P1 to another node P2 exists, if there is a fork call in the process P1 creating the process P2. A join edge from node P1 to node P2 indicates that P2 waits for P1 to terminate. A special entry node representing the beginning of the program and one or more end nodes indicating the termination of the program are also constructed. The entry node and end nodes are hypothetical processes represented in the graph to help make the formalism more intuitive. Please note that as far as the algorithm for computation of slices is concerned, fork and join edges are treated the same way. So, from now on we will use the term form edge to indicate both fork and join edges. The process graph for the example C-program given in Fig. 7.2(a) is shown in Fig. 7.2(b). In this example, the main program creates a child process and then waits for the child process to terminate. We now define fork edge, and join edge more formally.

**Definition 7.2 (Fork Edge)** A fork edge from a node P1 to a node P2 in a process graph exists if the last statement in the statement sequence represented by P1 is a fork call and P2 represents the statement sequence to be executed by the parent process immediately after this fork call or by the child process created by this fork call.

**Definition 7.3 (Join Edge)** A join edge from a node P2 to a node P1 in a process graph exists if all the following hold:

1. The statement sequence represented by P1 contains a wait call.
2. The statement sequence represented by P2 does not contain any fork call.
3. PO is the first predecessor node of both P1 and P2 which represents a statement sequence where the last statement is a fork call.

Figure 7.2 (a) An Example Program (b) The Process Graph and (c) The Concurrency Graph

7.2.2 Concurrency Graph

A process graph captures only the basic process structure of a program. This has to be extended to capture other Unix programming mechanisms such as interprocess communication and synchronization. We
achieve this by constructing a concurrency graph. A concurrency graph is a refinement of a process graph where the process nodes of the process graph containing message passing statements are split up into three different kinds of nodes, namely send node, receive node, and statement node. The significance of these nodes and the construction procedure of these nodes are explained as follows.

**Send node**: A send node consists of a sequence of statements which ends with a `msgsend` statement.

**Receive node**: A receive node consists of a sequence of statements which begins with a `msgrecv` statement.

**Statement node**: A statement node consists of a sequence of statements without any message passing statement.

To achieve node splitting, the code belonging to the process nodes of the process graph are examined. The splitting of a node depends on the sequence in which `msgsend()`/`msgrecv()` statements are present in a process node. All statements from the first statement of process node till the first `msgsend()` statement constitute a send node if there are no intervening `msgrecv()` statements. All statements between two consecutive `msgsend()` statements (including the later `msgsend()` statement) are represented as a send node. If a `msgsend()` statement is followed by a `msgrecv()` statement, then all statements between `msgsend()` and `msgrecv()` constitute a statement node. All statements after (and including) a `msgrecv()` statement till the next `msgsend()`/`msgrecv()` statement constitute a receive node. All statements from the beginning of the process node till the statement prior to `msgrecv()` constitute a statement node. All
statements after a msgsend() statement till the last statement of the process constitute a statement node if there is no intervening message passing statements.

Different nodes thus created are connected by appropriate control edges to show the control flow relationship among the created nodes. It can be observed that a send node may contain just a single statement - a msgsend() statement - if a sequence of statements along with a msgsend() statement follows a msgrecv() statement or there are two consecutive msgsend() statements. Similarly, a receive node may contain a single msgrecv() statement, if this statement is immediately followed by any message passing statement.

Interprocess communication due to message passings are represented by communication edges.

Definitions 7.4 (Communication Edge) A communication edge from a send node S to a receive node R in a concurrency graph exists if both the msgsend statement in the send node S and the msgrecv statement in the receive node R uses the same message queue.

Communication edges from every send node to the corresponding receive node is constructed indicating the partially blocking semantics of message passing. That is, if the 'msgrecv' statement in a 'receive' node R uses the same message queue 'msgqueue' which is used by the 'msgsend' statement in some 'send' node S, then a communication edge is
constructed from the node S to the node R. It is not possible to determine statically which msgsend statement exactly corresponds to which msgrecv statement. So, we have to consider all potential communication edges. That is if 'receive' nodes R1, R2 and 'send' nodes S1, S2 all use the same message queue, then communication edges exist from S1 to both R1 and R2 and also from S2 to both R1 and R2. Out of these four edges, only two will execute at runtime and the remaining two are potential which may not be required.

**Definition 7.5 (Concurrency Graph)** A concurrency graph $G_c = (N_c, E_c, f_c)$ is a directed graph where $N_c$ is a set of nodes referred as components. $E_c \subseteq N_c \times N_c$ is a set of edges and $f_c$ is a function assigning statements to components. Edges of a concurrency graph can be of following types: fork edges, communication edges, and control edges.

### 7.2.3 Concurrent Components

Each node of the concurrency graph is called a component. A component and a process may not be the same. In fact, a process might consist of more than one component. For example, in Fig.4.2©, the send node (which is the only send node in this example and is the first component found by splitting process node P1) and its successor node each represent a component although the two together represent a process. A component is the basic unit of concurrent execution. The maximal set of components which are capable of concurrent execution is called a concurrent set of components or just a concurrent component. Determination of shared dependence (described in the next section)
can not be done from a simple analysis of the source code. To determine shared dependence across two components one must know whether these two components are concurrent or not.

In order to determine the concurrent components in a concurrency graph, we need to determine all other components which are concurrent to a component. We present here an algorithm based on graph reachability for determining the set of components concurrent to a given component. $\text{Pc}_i$ represents the set of nodes of the concurrency graph which are concurrent to the node $i$. $\text{PC}$ is the set of all concurrent components of the concurrency graph. Formally, $\text{PC} = \{\text{PC}_1, \text{PC}_2, \ldots, \text{PC}_n\}$, where $n$ is the total number of components in the concurrency graph.

**Algorithm For Determination Of Concurrent Components**

**Input:** Concurrency graph, $(N_c, E_c)$  
**Output:** $\text{PC} = \{\text{PC}_1,\text{PC}_2,\ldots,|N_c|\}$.  
$\text{PC}_i = \{x \mid x \in N_c \text{ and } x \text{ is concurrent to node } i\}$  
for every node $n \in N_c$ do  
begin  
Construct the set, $\text{PU} = \{j \mid N_c \text{ and node } n \text{ is reachable from node } j \text{ traversing transitively along fork/control/communication edges}\}$  
Construct the set, $\text{PD} = \{k \mid k \in N_c \text{ and node } k \text{ is reachable from node } j \text{ traversing transitively along fork/control/communication edges}\}$  
Construct $\text{PC}_n = N_c - (\text{PU} \star \text{PD} \star \{n\})$  
End

To construct $\text{PC}_n$ for a node $n$, we may in the worst case have to traverse along all the edges of the concurrency graph. The complexity of this step is $O(E)$, $E$ being the number of edges in the concurrency graph. To construct
PC, we have to construct $PC_n$ for every node $n \in N_c$. Hence, the complexity of the algorithm to compute all concurrent components is $O(N \times E)$, where $N$ is the number of nodes and $E$ is the number of edges in the concurrency graph.

### 7.3 Concurrent Program Dependence Graph

A concurrency graph captures the dependencies among different components arising due to message passing communications among them. However, components may also interact through other forms of communication such as using shared variables. Access to shared variables may either be unsynchronized or synchronized using semaphores. Further, to compute a slice, in addition to representing concurrency and interprocess communication aspects, we need to represent all traditional (sequential) program instructions. To do this, we extend the concurrency graph to construct a third level graph called Concurrent Program Dependence Graph (CPDG). But, before we can construct CPDG by extending the concurrency graph, we need to resolve some problems arising due to the use of shared variables.

#### 7.3.1 Shared Dependence

In the program shown in Fig. 7.2(a), two processes P1 and P2 are created which execute in parallel and assign different values to the shared variables $x$. When the statements are executed in order $< 14, 15, 16, 17 >$, the value of $x$ at statement 17 depends on that at statement 15. But if the statements are executed in the order $< 14, 16, 17, 15 >$ the value of $x$ at statement 17 depends on which statement executed later. Similarly, the value of $x$ at statement 15 may either depend on that was assigned to it at statements 17 or 18 that at statement
depending on the order of execution of statements. However, in general the exact order of execution of the statements belonging to different concurrent components cannot be predicted beforehand. We handle such nondeterministic access to unsynchronized shared variables through construction of a shared dependence edges.

**Definition 7.6 (Shared Dependence Edge)** A shared dependence edge from a statement \( s_1 \) to another statement \( s_2 \) exists if the following hold:

1. \( x \in \text{def}(s_1) \),
2. \( x \in \text{ref}(s_2) \),
3. \( s_1 \) and \( s_2 \) can execute concurrently.

Please note that a shared dependence edge can exist only between two statements which belong to concurrent components. For the components which cannot execute concurrently, handling of the access to shared variables is deterministic and straightforward - we do not have to construct any shared dependence edges in such cases. After determining the concurrent components, we can find the shared dependence by associating the use of shared variable in one components to the definitions of the shared variables in the other components. For example, we find that there is a shared dependence from statement 17 to 15, and vice versa. This is represented by adding shared dependence edges from statement 17 to 15 and vice versa.
Let P1 be a send node and P2 be a receive node in a concurrency graph and communication edge exists from P1 to P2. For the sake of convenience we use the notation $s \in P$ to mean that $s$ is a statement in the statement sequence represented by the node P. Let a statement $s_1 \in P_2$ and $x \in \text{ref}(s_1)$, where $x$ is a shared variable. Also, let statements $s_2, s_3 \in P_1$ and both $s_2$ and $s_3$ define $x$. Now, a data dependence edge from one of these two statements to the statement $s_1$ in P2 might exist because only one but not both of these two might be a reaching definition of $x$ at $s_1$. That is, either $(x,s_2) \in \text{rd}(s_1)$, where $x$ is a shared variable. Also, let statements $s_2, s_3 \in P_1$ and both $s_1$ and $s_3$ define $x$. Now, a data dependence edge from one of these two statements to the statement $s_1$ in P2 might exist because only one but not both of these two might be a reaching definition of $x$ at $s_1$. That is, either $(x,s_2) \in \text{rd}(s_1)$ or $(x,s_3) \in \text{rd}(s_1)$. In this case, we do not construct shared dependence edges from both $s_2$ and $s_3$ to $s_1$ since components P1 and P2 are not concurrent. This means that reaching definition may cross process boundaries and in such cases we have to consider the communication edges in computing the reaching definitions. For example, consider the CFGs of P1 and P2. Let there be a path in the CFG of P1 from $s_2$ to the msgsend statement and $x$ is not defined by any node in that path except at $s_2$. Similarly, let there be a path in the CFG of P2 from the msgrecv statement to $s_1$ and $x$ is not defined by any node in that path. Considering the communication edge from the send node P1 to the receive node P2, we observe that $(x,s_2) \in \text{rd}(s_1)$ (i.e., definition of shared variable $x$ at $s_2$ is a reaching definition at $s_1$). Hence, a data dependence edge exists in this case from $s_2$ at P1 to $s_1$ at P2.
7.3.2 Semaphore Dependence

Semaphores are synchronization primitives which can be used to control access to shared variables. In the Unix environment, semaphores can be created by using the semget() call. The value of a semaphore can be set by the semct1() call. The increment and decrement operations on semaphores are carried out by the semop() call [63]. However, for simplicity we use the notations P(sem) and V(sem) to denote the semaphore decrement and increment operations respectively. Slicing of programs having synchronizations through semaphore operations is similar to programs having synchronizations through semaphore operations is similar to programs allowing unsynchronized access to shared variables within a single (P-V)-block, the access are serialized. A semaphore dependence indicates which P(sem) operations depend on which V(sem) operations across component boundaries.

Consider that the accesses of a shared variable x is synchronized using semaphore sem by various process. Let a component C1 in the concurrency graph uses this variables within a (P-V)-block without defining it within the block. Let another component C2, which is concurrent to C1, defines this variable within a (P-V)-block. The value of the variable x in C1 would be the last definition within the (p-V)-block in C2 (in case there are multiple definitions of s in this block). So, we need to construct a shared dependence edge from this last definition of x to the use of x in component C1.

7.3.3 Construction of CPDG
The first step in constructing a CPDG is to examine the process graph to represent data/control dependencies among individual statements belonging to every process node. The following nodes are constructed in the CPDG to achieve this: A process start node (which is a dummy node and does not represent any statement) for every process node of the process graph. Process start nodes are used to keep track of the process numbers, which helps us to determine the process number of any arbitrary statement in the graph. Individual statement nodes which represent different statements. The CPDG contains the following types of edges: data dependence edges, control dependence edges, shared dependence edges, communication edges, and fork edges.

The source of each control dependence edge is either a statement node representing a predicate, a wait call, or a process start node. All the statements in a process node which immediately follow a wait call are said to be controlled by the node W. We now construct a control dependence subgraph for each statement sequence representing a process node in the process graph. Informally, in each of these control dependence subgraphs, a control dependence edge from a node x to a node y exists, if any one of the following holds:

- x is a process start node and y is a statement which is not nested within any loops or conditional statements and is also not controlled by a wait call.
- x is a predicate and y is a statement which is immediately nested within this predicate.
• x is a wait call and y is a statement which is controlled by this wait call and is not immediately nested within any loop or conditional statement.

Data dependence edges between statements are constructed in the usual way i.e., computing reaching definitions. If a statement uses a shared variable, then a shared dependence edge may or may not be constructed depending on whether they belong to same concurrent components or not (as discussed in the last section). Fork edges existing between two process nodes in the process graph are now drawn from the individual statement nodes representing the fork statement to the corresponding process start nodes.

We present an algorithm Construct CPDG for construction of CPDG of a given concurrent program from its process graph and concurrency graph. We use the notation $s^p$ to refer to a statement belonging to a process node of a process graph $G_p$. Let $G_c$ be the concurrency graph corresponding to $G_p$. It can easily be observed that for every statement belonging to a process node of a process graph $G_p$, there is a corresponding statement belonging to a component in $G_c$ and vice versa. Let $s^c$ denote the statement in a component $C$ in $G_c$ corresponding to the statement $s^p$ belonging to a process node in $G_p$. The type of a statement $s$ is denoted as $s$. type.

Construct_CPDG

for all $n_p \in N_p$ do
begin
/* Examine all the statements in $n_p$ sequentially */
for all statements in $n_p$ do
begin

SLICING OF CONCURRENT PROGRAMS

\[ s^o \leftarrow \text{current statement examined} \]

/* Let \( s^c \) be the statement in \( G_c \) corresponding to \( s^o \) in \( G_p \) */
/* Let \( C \) be the component in \( G_c \) to which \( s^c \) belongs */

Construct all data and control dependence edges to \( s^o \) (if any)

if \( s^o \). Type = msgsend

\[ \text{Find } C' \text{ in } G_{cc} \text{ such that there exists an edge } (C, C') \in E_c \]
\[ s^c_1 \leftarrow \text{first statement in } C' /* By definition, } s^c_1 \text{ type = msgrecvg */} \]

Construct communication edge \((s^o, s^p_1)\)

end

else if \( s^o \). Type = fork

Add fork edges from \( s^o \) to process start nodes of the successors of \( n_p \)

e else if \( s^o \). Type = shared variable read

begin

/* Let \( x \) is the shared variable in ref \((s^o)\) */

\[ CC \leftarrow \text{concurrent component of } C \]
\[ C_x \leftarrow \{C_i | C_i \in CC \land \exists \text{ a statement } t \in C_i \text{ such that } x \in \text{def}(t)\} \]

for all \( C_i \in C_x \) do

begin

Find \( s^c_k \) such that \((s^c_k, C_i \land x \in \text{def}(s^c_k))\)

Construct shared dependence edge \((s^o_k, s^p)\)

end

end

else if \( s^o \). Type = shared variable read /* within a \((P0V)\)-block */

if \( x \) is not defined within the \((P-V)\)-block before \( s^o \)

begin

\[ CC \leftarrow \text{concurrent component of } C \]
\[ C_x \leftarrow \{C_i \in CC \land V(\text{sem}) \land C_j\} \]

for all \( C_i \in C_x \) do

begin

Find \( s^c_k \) in \( C_i \) which precedes \( V(\text{sem}) \) such that \( x \in \text{def}(s^c_k) \)

Construct shared dependence edge \((s^o_k, s^p)\)

end

end
The CPDG for the example program of Fig. 7.2(a) is shown in Fig. 7.3. The statement numbers represented by a node are shown inside every node of the CPDG. We can observe that normal data dependence edges exist from statement 6 to 7, 1 to 6, etc. Shared dependence edges exist from statement 6 to 17, 15 to 18, etc. Please note that no shared dependence edge exists between the statements 6 and 15 because the components to which these two statements belong are not concurrent.

Figure 7.3: The CPDG for the Example Program Shown in Figure 7.2(a)
7.4 Implementation of Our Static Slicing Methodology

We have designed and implemented a static slicing tool which we have named Statif slicer for Concurrent Programs (SSCP), based on the method discussed in earlier sections. The main motivation for our implementation is to experiment with the usefulness and accuracy of our slicing methodology rather than to develop a commercial product.

7.4.1 Slicing Criterion

We define the slicing criterion for a concurrent program to be a triplet $< P, s, V >$, where $P$ is a process identifier, $s$ is a statement in the process, and $V$ is a set of variables. It is necessary to specify the process number in the slicing criterion, because the same source line may belong to different processes according to the Unix process creation semantics. Once the CPDG has been constructed, the problem of determining the slice becomes the graph reachability problem in the CPDG. All statements reachable from the node specified in the slicing criterion by traversing transitively along the edges (all types of edges) of the CPDG are included in the slice. We can summarize these slicing steps as follows:

- Construct the process graph of the given program.
- Construct the concurrency graph referring to the process graph and the source code.
- Determine the concurrent components.
- Construct the CPDG of the program using the algorithm Construct_CPDG.
- Input slicing criterion $< P, s, V >$ and map $s$ to the proper node in the CPDG.
Compute the slice using the reachability analysis.

### 7.4.2 Module Structure of SSCP

The schematic design of SSCP is shown in Fig. 7.4. The arrows among the modules represent flow of information. The important components of SSCP and their roles are as follows.

The Program analyzer analyzes the program code to extract all information pertaining to process creation, join and termination. We have used the standard UNIX tools `lex` and `yacc` to carry out the syntax and semantic analysis of the program code. Other information collected for every statement during this analysis are: type of the statement (e.g. assignment, I/O, message passing, etc.), type of variables used in the statement (e.g. shared or ordinary), variables used and defined in the statement, and process number to which it belongs.

![Figure 7.4: Schematic Design of the Static Slicer.](image-url)
The **Process graph constructor** uses the information collected by the program analyzer component to create the process graph. The procedure used for construction of the process graph is already described in Section 7.2.1. We have used data structures linked through pointers to implement the edges and nodes of the process graph.

The **Concurrency graph constructor** refers to the process graph generated by the process graph constructor as well as the information generated by the program analyzer to extract the send/receive calls made by various processes. Using this information it splits the relevant nodes of the process graph into statement nodes, send nodes and receive nodes. Communication edges are constructed between corresponding send and receive nodes. The concurrency graph constructor component also incorporates the algorithm to compute the concurrent components.

The inputs to the **CPDG constructor** are the source code, concurrency graph, process graph, and the information regarding the concurrent components. It analyzes the source code and adds all the data, control shared, communication dependence and fork edges among the relevant statement nodes belonging to every process node of the process graph.

The **Slicer** implements the slicing algorithm and applies the graph reachability criteria on the CPDG. Input to the slicer is the CPDG and the slicing criterion.
Design of the Graphical User Interface (GUI) for viewing the slice and specifying the slicing criterion for a concurrent / distributed program is much more compiled than that for sequential programs. Due to the complex nature of large concurrent programs it is necessary to make the slicing information as meaningful to the user as possible. The user selection of slicing criterion and the display of the resultant slice should be done carefully. The GUI using X/MOTIF has been developed keeping these points in mind. The user has the option of viewing any of the three levels of representation of the program. The user can enter the slicing criterion by examining the process graph, concurrency graph, or the code contained in any of these, or the full code of the program. By clicking on a node in any of the graph representations, the user can see the code it represents. Slicing information is displayed at three hierarchical levels for better understanding. The sliced process graph is presented graphically to indicate the processes in the slice. The sliced concurrency graph is presented to indicate the interprocess communications and synchronizations relevant to the slice. The user can click on any node of the graph representation to view the slice of that node with respect to the specified slicing criteria. It is possible that the slice of a node may be empty.

The tool supports the options LOAD, LEVEL1, LEVEL2, and SLICE. The LOAD option is used for loading different program files. The program code of the program is displayed as soon it is loaded. By clicking a LEVEL1 button one can see the process graph of the loaded program. The GUI screen for the process graph of a sample program is shown in Fig. 7.5 and the source code
of the program itself is shown in Fig. 476. Selection of the options LEVEL2 and SLICE lets one view the concurrency graph and slice of the program respectively.

The display screens do not display all information regarding the graphs for clarity and simplicity. For example, the entry node and the end nodes are not displayed. Process node number and the numbers of statements that belong to a process node are displayed by the side of that node. By clicking on a process node of the process graph the user can view the statements belonging to that node.

The user can select the option SLICE from the tool bar to compute a slice. The user is prompted to enter the slicing criteria. The slicing criteria is entered by clicking on the required process node of the process graph and then entering the statement number and the required variable. The slice can be viewed at different levels. Fig. 7.7 shows a slice of the whole program with respect to the slicing criteria < 8,27, b>. The statements belonging to the slice are highlighted. Two more options are provided to view the sliced process graph and sliced concurrency graph.

The user can also view the different components concurrent to a given component by choosing the option and clicking on the given component. Fig. 7.8 displays the concurrent components of component 8 of the given concurrency graph. We can easily observe that Fig. 7.5 to Fig. 7.8 give a better understanding of programs since a program can be observed in increasing levels of details. This becomes especially true if the programs are large or
complex. The user can view the various processes involved and the code belonging to each of these processes. Concurrency graphs also indicate how different processes communicate with each other.

7.5 Related Work

Cheng proposed a representation for concurrent programs where he generalized the notion of CFG and PDG [63]. His representation, called Program Dependence Net (PDN), contains edges for selection, synchronization, and communication dependence in addition to data and control dependence edges. Static slices are computed by solving a reachability problem in PDN. But the semantics of synchronization and communication dependences are not considered. Cheng's method is applicable to programs written in Occam-2 like languages. He does not consider interprocess communication using shared variables.

Krinke proposed a method for static slicing of threaded programs [78]. He extended the structures of CFG and PDF for threaded programs with interference. Interference is defined as data flow which is introduced through use of variables which are common to parallel executing statements. He has not considered interprocess communication using message passings. In contrast to this, we have considered slicing of multi-process programs and have handled interprocess communication using both shared variables and message passings.
We have not come across any work focusing on slicing of concurrent programs in Unix process environment. An interesting aspect of the Unix process environment is the support of different interprocess communication techniques such as message passing and shared memory. Ordering of messages arriving in a message queue is not possible except for the case when these are sent from a single process. Therefore, we have considered a conservative approach to match a msgsend statement with the corresponding msgrecv statement as discussed in Section 7.2.2. Ordering of accesses of shared variables by different processes is again not possible. Hence, potential data dependence exists from every definition of a shared variable in a process to some reference of that variable in some other process. We reduce the number and correctly construct such dependences by concurrency analysis as discussed in Sections 7.3.1 and 7.3.2.

Our intermediate graph representation is substantially different from those of Cheng's and Krinke's. Our representation is a hierarchical one which allows one to construct it incrementally through three levels. We can correctly construct the shared dependence edges using the information of concurrent components. The GUI that we have designed leads one to better understanding of concurrent programs since slicing at different levels of abstraction is possible.
SLICING OF CONCURRENT PROGRAMS

Figure 7.5: Process Graph for the Sample Program Shown in Figure 7.6

0 : main() {
1 : shm a; /* Shared memory declaration */
2 : shm b; /* Shared memory declaration */
Slicing of Concurrent Programs

```c
3    : msgget ms1; /* Messageget declaration */
4    : msgget ms2; /* Messageget declaration */
5    :     if (fork() != 0) {
6        :         a = b;
7        :         msgsend (ms1, data);
8        :         a = 2;
9        :         msgsend (ms2, data);
10       :         c = 1;
11       :         b = a+b;
12       :     if (fork() != 0) {
13           :         a = 1;
14           :         wait(0); }
15     :   else {
16       :       a = 2;
17       :   }
18       :   a = 3; }
19     :   else {
20       :     msgrecv (ms1, data);
21       :     c = 1;
22       :     a = 5;
23       :     b = 2;
24       :     c = 3;
25       :     a = b+c
26       :     b = c+a
27       :     c = a+b;
28       :     if (fork() != 0) {
29           :         a = 1;
30           :         msgrecv(ms2, data);
31       :     wait(0); }
32     :   else
33       :     a = 1;
34   }
35 }```

CHAPTER 7 A PROCESS MODEL AND ITS HIERARCHICAL REFINEMENTS


```
main() {
    shm a; /* Shared memory declaration */
    shm b; /* Shared memory declaration */
    msgget ms1; /* Messageget declaration */
    msgget ms2; /* Messageget declaration */
    if (fork() != 0) {
        a = b;
        msgsend (ms1, data);
        a = 2;
        msgsend (ms2, data);
        c = 1;
        b = a+b;
        if (fork() != 0) {
            a = 1;
            wait();
        } else {
            a = 2;
            a = 3;
        }
        msgrecv (ms1, data);
        c = 1;
        a = 5;
        b = 2;
        c = 3;
        a = b+c;
        b = c+a;
        c = a+b;
        if (fork() != 0) {
            a = 1;
            msgrecv(ms2, data);
            wait();
        } else {
            a = 1;
        }
    } else {
        msgrecv (ms1, data);
        c = 1;
        a = 5;
        b = 2;
        c = 3;
        a = b+c;
        b = c+a;
        c = a+b;
        if (fork() != 0) {
            a = 1;
            msgrecv(ms2, data);
            wait();
        } else {
            a = 1;
        }
    }
}
```

Figure 7.6: The GUI Screen Along with A Sample Program
CHAPTER 7 A PROCESS MODEL AND ITS HIERARCHICAL REFINEMENTS
SLICING OF CONCURRENT PROGRAMS

```c
7     :     msgsend (ms1, data);
8     :     a = 2;
9     :     msgsend (ms2, data);
10    :     c = 1;
11    :     b = a + b;
12    :     if(fork() !=0) {
13    :       a=1;
14    :       wait(0); }
15    :     else {
16    :       a=2;
17    :     }
18    :     a=3; }
19    :     else {
20    :     msgrecv (ms1, data);
21    :     c=1;
22    :     a=5;
23    :     b=2;
24    :     c=3;
25    :     a = b+c
26    :     b = c+a
27    :     c = a+b;
28    :     if (fork() !=0) {
29    :       a=1;
30    :     msgrecv(ms2,data);
31    :     wait(0); }
32    :     else
33    :     a=1;
34    :     }
35    : }
36    : /* End main */
```
Figure 7.7: Slice of the Sample Program w.r.t. the Criterion < 8,27,b>
Figure 7.8: The Concurrency Graph of the Sample Program

7.6 Summary

We have proposed a hierarchical graph representation of concurrent programs in Unix process environment. The three hierarchical levels of this graph
representation are process graph, concurrency graph, and CPDG. This hierarchical representation not only lets us efficiently calculate static slices but also facilitates in understanding the behaviour of concurrent programs. We have considered interprocess communication using both shared variables and message passing. We handled complications due to message queues and access to unsynchronized data. The later was resolved through a concurrency analysis of the concurrency graph. Concurrent components are computed from the concurrency graph which are later used in constructing the shared dependence edges in CPDG. This method eliminates most of the potential shared dependence edges from the graph and helps improving the accuracy of computed slices. Our method cannot handle dynamically created processes, but dynamically created process is beyond the scope of any static slicing method. The prototype tools that we have developed validates our proposed methodology. We have tested our slicing tool using several examples.