A Unifying Theory of Correct Concurrent Executions[*]

Banu Özden
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-1084

Avi Silberschatz
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

An ideal system is one that performs program operations in the order specified by the program and executes atomic program segments exclusively. Although this system model simplifies the task of reasoning about both sequential and concurrent programs, its straightforward implementation yields poor performance. To enhance performance, concurrency and pipelining techniques can be used, which may result in data accesses that are performed in an order which is different from the order specified by the program, which may result in incorrect executions. An execution is correct if its result is equivalent to the result that could have been obtained had the execution taken place on the ideal system. In this paper, we develop a unified general theory of correct executions where the access orders differ from the access order on the ideal system. Our unifying theory is applicable to a variety of programming paradigms, application domains, and architectures. It provides a verification tool to test the correctness of an execution, and allows us to devise more efficient protocols for various systems.

Index Terms

Access order, concurrency, concurrent execution, concurrent programming, correctness, databases, distributed systems, multiprocessors, pipelining, sequential consistency, serializability, synchronization.

1 Introduction

In order to aid the programmers with the task of reasoning about the correctness of their programs, an *execution model* is usually provided, which is a description of the execution order of the various operations of a program. Examples of execution models are *sequentiality* for sequential systems [1], *sequential consistency* for multiprocessors [1], and *serializability* for database systems [2].

Although the availability of an execution model simplifies the reasoning about the programs, additional synchronization constructs must be available so that programmers can explicitly express a specific order on the execution of the operations of their programs. Examples of synchronization constructs are semaphores, critical sections, monitors, barrier and condition synchronization primitives [3, 4, 5].

[*]This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant Nos. IRI-9003341 and IRI-9106450, and grants from the IBM and Hewlett-Packard corporations.

There is a basic simple protocol to implement each of these execution models and a known effective implementation of each of these synchronization constructs. For instance, sequentiality can be ensured by executing the operations of a sequential program in the program order, sequential consistency can be ensured by executing the operations of each program of a concurrent program in the program order, and serializability can be ensured by executing transactions in a serial order.

Since these protocols and implementations, in general, yield poor performance, a significant amount of research has been done to devise methods to obtain better performance. The two most common techniques for achieving this are pipelining and concurrency [1, 2, 6, 7, 8, 9, 10, 11]. Pipelining is a method for overlapping the execution of multiple operations of a process, whereas concurrency is a method for overlapping the execution of multiple operations of different processes (a process is an execution of a sequential program, namely, an execution of a sequential operation stream.) The use of concurrency and pipelining must be controlled, since both may change the order in which data accesses are performed, and therefore may yield incorrect execution.

Pipelining allows an operation to be issued, before the previous operations in the program order are issued or performed. In order to mask latency of interconnection networks, memory accesses are issued before the previous accesses are performed in some shared memory multiprocessors, and messages are sent before the previous messages are delivered in some distributed memory multiprocessors. If the interconnection network is asynchronous, then the order, in which memory accesses are executed and messages delivered, may differ from the order specified by the program. Similarly in a pipelined processor, in order to increase the throughput, an instruction can be issued before completion of a previous instruction, which may cause memory accesses to be performed in an unintended order.

Concurrency allows several processes to execute simultaneously. Typically, programs require more than one data item to be accessed atomically (without interleaving with other's data accesses). Examples are programming languages in which sequence of statements can be specified as atomic, or databases where each transaction should be atomic. Since executing the atomic sections in isolation may degrade the performance, concurrent executions are allowed, which can result in incorrect interleavings of data accesses.

The problem that the execution order of data accesses of a program can be different from the intended order and, therefore, the execution may be incorrect exists in numerous programming paradigms, application domains, and architectures. Examples are sequential programming, concurrent programming based on shared data or message-passing, parallel programming, centralized and distributed databases, single processor systems, and shared and distributed multiprocessor systems. Although the nature of the problem is the same, there has been no research present a unified solution. It is the aim of this paper to develop a unifying theory for correct execution.

A system that performs operations in the program order and executes atomic sections exclusively is referred to as an *ideal system*. We assume that each program is correct in the sense that if it were executed on an ideal system, then its result is the desired one. It is the responsibility of the programmers to ensure that their programs are indeed correct. We refer to an execution on the ideal system as the *specification of the correct* *execution*, or the *correctness criterion.* We refer to an execution whose result is equivalent to the result of the execution on the ideal system, as a *correct execution.*

Given a specification of a correct execution, our goal is to define the class of correct executions whose access order is less restrictive than the the access order of the corresponding execution on the ideal system. We develop a general theory of correct execution that is applicable to any correctness criterion that can be expressed as follows. An execution on an ideal system is a set of sequential processes, each of which is a sequence of atomic actions. An atomic action is a sequence of indivisible data accesses. The order among the atomic actions of different processes can be expressed with a partial order. The processes can run in parallel and can share data. This correctness criterion is sufficiently general to encampus sequentiality, sequential consistency and serializability as special cases.

Our unifying theory provides a verification tool to test the correctness of an execution. It will have impact on understanding the access ordering problem, and will allow us to devise more efficient protocols for various systems. Furthermore, the unified theory will allow one to extend results developed in one type of system to other systems.

The remainder of this paper is organized as follows. In Section 2, we present examples for access ordering problem. In Section 3, we introduce the system model. In Section 4, we discuss the differences between correctness, sequential consistency and serializability. In Section 5, we introduce various classes of correctness. In Section 6, we present the concepts of hierarchical graphs and hierarchical polygraphs to reduce the complexity of testing algorithms, whereas in Section 7, we develop testing algorithms for different classes of correctness. We present our conclusions in Section 8, and prove the theorems in the Appendix.

2 Examples

In order to motivate our work, we will give examples from several programming and application paradigms and different architectures where concurrency and pipelining change the execution order with respect to the intended order.

Consider a single pipelined processor system, which allows the issuing of memory accesses of the next operation, before the execution of previous instructions are completed. Suppose that sequential program $K1$ in Figure 1 is executed on such a processor. In this case, it is possible that operand a of the second instruction is loaded before the value of a is calculated and stored by the first instruction. Hence, the execution will be incorrect. On the other hand, if the operands of the third instruction are loaded before the previous instruction is completed, the result will be correct.

Consider Peterson's solution to two-process critical section problem as shown in Figure 2. The program is written with the assumption that the system is sequentially consistent. Suppose this program is executed on a shared memory multiprocessor. If the basic load and store operations are indivisible, and the system does not pipeline the memory accesses, then the execution of this program will yield correct result, namely, at most one process can be in the critical section. Now, suppose that the system allows pipelining of loads and stores.

```
a := b/c;
d := d + a;
e := e − f;
```

Figure 1: Sequential program $K1$.

$P0$	$P1$
shared $F0, F1$: **boelean**;	**shared** $F0, F1$: **boelean**;
shared $turn, x, y$: **integer**;	**shared** $turn, x, y$: **integer**;
$F0 := true$;	$F1 := true$;
$turn := 1$;	$turn := 0$;
while($F1$ **and** $turn = 1$) **do skip**;	**while**($F0$ **and** $turn = 0$) **do skip**;
Critical Section	Critical Section
$F0 := false$;	$F1 := false$;

Figure 2: $K2$: Peterson's solution to two-process critical section problem.

Then, the following order of events is possible. Suppose that initially $F0 = F1 = false$. Process P_0 issues the requests to store the value $true$ in $F0$ and the value 1 in $turn$. Following this, it issues the requests to load $F1$ and $turn$, and then enters the critical section. Process P_1 issues the requests to store the value $true$ in $F1$ and the value 0 in $turn$. Following this, it issues the requests to load $F0$ and $turn$. The request from process P_0 to store $true$ into $F0$ is still not performed, and this load request of process P_1 returns $false$ as the value of $F0$, and process P_1 enters the critical section. Hence, the execution is incorrect.

Consider a client-server system, in which each server manages a set of data, and clients send read and write requests to the appropriate server to access and update data. Suppose that the client programs in Figure 3 are written with the assumption that the system is sequentially consistent. Servers $S0$ and $S1$ manage objects $object0$ and $object1$ respectively. Each client reads two objects from two different servers, caches the objects into local buffers, increments each word of the objects, and updates the copies of the objects in the servers. Sequential consistency can be ensured by waiting for an acknowledgement message from the server to which a request is sent, before another request is issued. Since such a protocol yields poor performance, the system may choose to pipeline the requests. In this case, the following execution is possible. Suppose that initially all the words of both objects are zero. After $C0$ sends the update request to $S0$, $C0$ sends read and then the update requests to $S1$. $C1$ sends a read request to $S1$, which arrives at $S1$ after the write request from $C0$. Therefore, $C1$ reads the value written by $C0$. After sending update message to $S1$, $C1$ sends read request to $S0$. This request arrives at $S0$ before the update request by $C0$. Hence, $C1$ reads the initial value of $object0$. This execution yields an incorrect final state in which all the words of $object0$ are one, and all the words of $object1$ are five, whereas a correct execution should result in the state in which all the words of $object0$ and

$C0$	$C1$
shared $object0, object1$;	**shared** $object0, object1$;
local $buffer, i, n$;	**local** $buffer, i, n$;
read$(object0, buffer, n)$;	**read**$(object1, buffer, n)$;
for $i = 0$ **to** $i < n$ **do**	**for** $i = 0$ **to** $i < n$ **do**
$buffer[i] := buffer[i] + 1$;	$buffer[i] := buffer[i] + 3$;
write$(object0, buffer, n)$;	**write**$(object1, buffer, n)$;
read$(object1, buffer, n)$;	**read**$(object0, buffer, n)$;
for $i = 0$ **to** $i < n$ **do**	**for** $i = 0$ **to** $i < n$ **do**
$buffer[i] := buffer[i] + 2$;	$buffer[i] := buffer[i] + 4$;
write$(object1, buffer, n)$;	**write**$(object0, buffer, n)$;

Figure 3: $K3$: Client programs $C0$ and $C1$.

$P1$	$P2$
$x := x + 10$;	$x := x + 100$;
$y := y + 10$;	$y := y + 100$;

Figure 4: Concurrent program $K4$.

$object1$ are five.

Consider concurrent program $K4$ depicted in Figure 4, which is written with the assumption that increment statements are atomic. If initially $x = 0$, then a correct execution must yield $x = 110$, which can be ensured by executing each statement atomically. However, such a system yield poor performance. To improve performance, the increment statement might be implemented as a sequence of three indivisible operations: (i) load a register with the value of x; (ii) add 10 or 100 to it; (iii) store the result in x. Thus, in the concurrent program above, the final value of x might be 10, 110, or 100. Concurrent execution of $P1$ and $P2$ must be synchronized to enforce restrictions on possible interleavings.

Consider a database system that is implemented on a distributed system with the client-server model. The transaction manager is the server and the transactions are clients that send read and write requests to the server. Transactions are written with the assumption that the system will ensure serializability. The system uses a concurrency control protocol which orders transactions, and allows a transaction to issue its operations, only if all the previous transactions in the order complete. If the system does not allow pipelining of data accesses, then this protocol ensures serial executions. However, if the system allows pipelining and interprocessor communication is asynchronous, this protocol may not ensure serializability. Suppose the protocol ordered transactions in Figure 5 such that $T0$ is to be executed before $T1$. Let us represent the chronological order in which instructions are executed in the system with a schedule. Figure 6 depicts the schedule generated by this protocol when the data accesses are not pipelined. The schedule is serial. Now suppose that the system

$T0$	$T1$
shared A, B;	**shared** A, B;
local $temp$;	**local** $temp0, temp1$;
read$(A, temp)$	**read**$(A, temp0)$
$temp := temp - 50$;	temp1:= temp0*0.1;
write$(A, temp)$;	$temp0 = temp0 - temp1$;
read$(B, temp)$;	**write**$(A, temp1)$;
$temp := temp + 50$;	**read**$(B, temp0)$;
write$(B, temp)$	$temp0 := temp0 + temp1$;
	write$(B, temp0)$;

Figure 5: $K5$: Two transactions $T0$ and $T1$.

allows pipelining. The following order of events is possible. After $T0$ issues all its read and write accesses, it commits. $T1$ issues *read*$(A, temp0)$. This request arrives the server before *write*$(A, temp)$ of $T0$. $T1$ reads the initial value of A. $T1$ issues *read*$(B, temp0)$ which arrives the server after *write*$(B, temp)$ of $T0$. Hence, $T1$ reads the value of B written by $T0$. The execution is not serializable.

The protocol above ensures serializability, if there is no pipelining. However, such a protocol decreases performance unnecessarily. Consider the execution in Figure 7. Although transactions are executed concurrently, the result is equal as if $T1$ is executed after $T0$. Now consider the execution in Figure 8. Transactions are executed concurrently, but the result is not equal to any serial execution of $T0$ and $T1$. Hence, any interleaving of operations may not yield correct result, execution of $T0$ and $T1$ must be synchronized to enforce restrictions on possible interleavings.

Consider parallel program $K6$ in Figure 9 in which barrier synchronization is used. The end sync construct specifies a barrier, which means that a process cannot execute the statements following the barrier before all other processes reach the barrier. The forall construct specifies that processes can execute the loop concurrently, and each iterate of the loop is executed by another process. Suppose that $N = 4$ and processes P_1, P_2, P_3 and P_4 execute the iterations 1,2,3 and 4 for both forall loops, respectively. Program $K7$ in Figure 10 illustrates the statements that each process will execute in this case. Note that there are execution orders of operations that are different than the order specified by the program, but yield correct execution. For example, the execution will be correct if P_1 only waits for P_2, before issuing load $a[2, 1]$ after the barrier.

Consider concurrent program $K8$ in Figure 11, which consists of a producer program and consumer program. The program uses semaphores to specify a specific execution order, namely, initially both buffers are empty and producer writes into $buffer0$ and $buffer1$, and the consumer can read the buffers only after the producer writes into the buffers, and the producer can write another item into the buffers only after the consumer reads the buffers. Although this implementation is correct, it may yield poor performance. Suppose this program

$T0$	$T1$
$\textbf{read}(A, temp)$	
$temp := temp - 50;$	
$\textbf{write}(A, temp);$	
$\textbf{read}(B, temp);$	
$temp := temp + 50;$	
$\textbf{write}(B, temp)$	
	$\textbf{read}(A, temp0)$
	temp1:= temp0*0.1;
	$temp0 = temp0 - temp1;$
	$\textbf{write}(B, temp1);$
	$\textbf{write}(B, temp1);$
	$\textbf{read}(B, temp0);$
	$temp0 := temp0 + temp1;$
	$\textbf{write}(B, temp0);$

Figure 6: A serial schedule of $T0$ and $T1$.

$T0$	$T1$
$\textbf{read}(A, temp)$	
$temp := temp - 50;$	
$\textbf{write}(A, temp);$	
	$\textbf{read}(A, temp0)$
	temp1:= temp0*0.1;
	$temp0 = temp0 - temp1;$
	$\textbf{write}(B, temp1);$
$\textbf{read}(B, temp);$	
$temp := temp + 50;$	
$\textbf{write}(B, temp)$	
	$\textbf{read}(B, temp0);$
	$temp0 := temp0 + temp1;$
	$\textbf{write}(B, temp0);$

Figure 7: A concurrent serializable schedule of $T0$ and $T1$.

$T0$	$T1$
read$(A, temp)$	
$temp := temp - 50;$	
	read$(A, temp0)$
	temp1:= temp0*0.1;
	$temp0 = temp0 - temp1;$
	write$(B, temp1);$
	read$(B, temp0);$
write$(A, temp);$	
read$(B, temp);$	
$temp := temp + 50;$	
write$(B, temp)$	
	$temp0 := temp0 + temp1;$
	write$(B, temp0);$

Figure 8: A concurrent nonserializable schedule of $T0$ and $T1$.

```
shared $a[N, N]$;
constant $N$;
local $i, j, k$;

forall($i = 1; i \leq N$) in parallel
  for($k = 0; \frac{N}{2}; k++$)
    for($j = 0; N; j := j + 2 * k$)
      a[i,j]:= a[i,j] + a[i,j+k];
end sync
forall($j = 1; j \leq N$) in parallel
  for($k = 0; \frac{N}{2}; k++$)
    for($i = 0; N; i := i + 2 * k$)
      a[i,j]:= a[i,j] + a[i+k,j];
end sync
```

Figure 9: Parallel program $K6$.

P_1	P_2	P_3	P_4
$a[1,1] := a[1,1] + a[1,2]$;	$a[2,1] := a[2,1] + a[2,2]$;	$a[3,1] := a[3,1] + a[3,2]$;	$a[4,1]$:=a[4,1]+$a[4,2]$;
$a[1,3] := a[1,3] + a[1,4]$;	$a[2,3] := a[2,3] + a[2,4]$;	$a[3,3] := a[3,3] + a[3,4]$;	$a[4,3]$:=a[4,4]+$a[4,4]$;
$a[1,1] := a[1,1] + a[1,3]$;	$a[2,1] := a[2,1] + a[2,3]$;	$a[3,1] := a[3,1] + a[3,3]$;	$a[4,1]$:=a[4,1]+$a[4,3]$;
barrier synchronization;	barrier synchronization;	barrier synchronization;	barrier synchronization;
$a[1,1] := a[1,1] + a[2,1]$;	$a[1,2] := a[1,2] + a[2,2]$;	$a[1,3] := a[1,3] + a[2,3]$;	$a[1,4] := a[1,4] + a[2,4]$;
$a[3,1] := a[3,1] + a[4,1]$;	$a[3,2] := a[3,2] + a[4,2]$;	$a[3,3] := a[3,3] + a[4,3]$;	$a[3,4] := a[3,4] + a[4,4]$;
$a[1,1] := a[1,1] + a[3,1]$;	$a[1,2] := a[1,2] + a[3,2]$;	$a[1,3] := a[1,3] + a[3,3]$;	$a[1,4] := a[1,4] + a[3,4]$;

Figure 10: Program $K7$.

Producer	Consumer
shared $buffer0, buffer1$;	**shared** $buffer0, buffer1$;
local $temp0, temp1$;	**local** $temp0, temp1$;
semaphore $full = 0, empty = 1$;	**semaphore** $full = 0, empty = 1$;
repeat	**repeat**
produce an item in $temp0$;	**wait**($full$);
produce an item in $temp1$;	**read**($buffer0, temp0$);
wait($empty$);	**read**($buffer1, temp1$);
write($buffer0, temp0$);	**signal**($empty$);
write($buffer1, temp1$);	consume the item in $temp0$;
signal($full$);	consume the item in $temp1$;
until$false$;	**until**$false$;

Figure 11: $K8$: Producer and consumer programs.

is executed on a sequential processor and the system uses a protocol that orders the accesses to shared data as shown in Figure 12. In this case, the execution will be correct. However, not all possible interleavings of reads and writes yield correct result. Execution of consumer and producer processes must be synchronized to enforce restrictions on possible interleavings.

Consider the execution of the program in Figure 11 on a shared or distributed memory multiprocessor. Suppose the system uses semaphores, but the system allows pipelining of data accesses. If the program is executed on a shared memory multiprocessor, where the interconnection network between processors and memory modules is asynchronous, and if it is executed on a distributed memory multiprocessor, where the interconnection network between processors is asynchronous, then the following order of events is possible. After the producer issues requests to write into both buffers, it issues a signal request on semaphore ull. The signal request is performed. The consumer issues a wait request on semaphore $full$ and issues read operations. Due to the asynchronous behavior of the interconnection network, the previous writes from the producer are

Producer Process	Consumer Process
$\mathbf{write}(buffer0, temp0);$	
	$\mathbf{read}(buffer0, temp0);$
$\mathbf{write}(buffer1, temp1);$	
	$\mathbf{read}(buffer1, temp1);$
$\mathbf{write}(buffer0, temp0);$	
	$\mathbf{read}(buffer0, temp0);$
$\vdots$	$\vdots$

Figure 12: An order of execution of reads and writes of programs $C0$ and $C1$ that results in correct execution.

still not performed. The read requests of the consumer are performed. The consumer reads incorrect values, and thus this execution is incorrect.

3 System Model

A concurrent execution involves a set of sequential *processes,* $\mathbf{P} = \{p_1, p_2, ..., p_n\}$ [*], and a set of non-overlapping data structures called *entities,* $\mathbf{E} = \{e_1, e_2, ..., e_m\}$. A process is the execution of a sequential program, which consists of a finite sequence of operations. Processes communicate with each other through shared entities. There is only one valid version of an entity at any time. This means that if there are several copies of an entity, these copies are kept coherent. Accesses to entities are indivisible, which means that the effect of performing a read or write on an entity is equivalent to the case where the read and write are executed exclusively. We allow the granularity of an entity to be larger than one memory word. Note that an entity is not necessarily shared.

This model encompasses *sequential* and *shared memory systems* in which a memory word is an entity, *client-server systems* in which shared data of any size accessed through a server process is an entity, *database systems* in which shared items are entities, and *message passing systems* in which a message buffer is an entity. For message passing systems, a send operation can be viewed as a write operation on the message buffer of the receiver processes, and a receive operation as a read operation on the message buffer.

A process may use *local data buffers* to cache entities. Local data buffers are not shared among processes. For example, registers in sequential and tightly coupled shared memory programming can be viewed as local data buffers. Similarly, in client-server systems or databases, the variables in the address space of a process, in which an entity is buffered, or from which an entity is updated can be viewed as local data buffers.

In this paper, we make a simplifying assumption that the accesses to local data buffers are executed in the program order. This assumption can be relaxed either by modeling the local data buffers as entities, or by

[*] $n \geq 1$. We allow that n to be one to be able to apply the theory also to sequential programs.

```
entities $x, y, z$ : integer;

$z := x + y$;
if $x \geq 10$ then
  $x := z - x$;
```

Figure 13: Program $K9$.

deriving a more sophisticated system model. The latter issue is a future research topic.

3.1 Issuing and Performing an Operation

In order to develop a comprehensive theory, we distinguish between the actions of *issuing an operation* and *performing an operation* on an entity. Issuing a read or a write operation means that a request to perform the operation is made, whereas performing a read or a write operation means that the requested operation is serviced. We say that a read is serviced at the moment when the value it will return is fixed. Similarly, a write is serviced at the moment when a subsequent read can return the value written.

We are only interested in *read* and *write* operations, denoted by R and W respectively. Hence, we use the term operation only to refer to read and write operations. We use the notations $R_i^j(e)$ and $W_i^j(e)$ to denote that if the operations were executed in the program order, jth operation of process P_i would be a read and write operations on entity e, respectively. When no confusion arises, we will omit the subscript or the superscript.

3.2 Program Order

To simplify the presentation, we sometimes refer to the graph representation of a relation R also as R, and to the underlying relation of a graph G as G. The *program order* of a concurrent execution specifies the order in which entity accesses would have been performed if they were executed in the order specified by the concurrent program. The program order IB_i for a process P_i is a total order on the set O_i of operations executed by process P_i. We also refer to IB_i as the schedule of process P_i. IB_i is analogous to the concept of trace defined in [9], and the concept of transaction in databases. To illustrate, Figure 14 displays the program order generated by executing program $K9$ in Figure 13, when initially $x \geq 10$, and Figure 16 displays the program order generated by executing program $K10$ in Figure 15. In these figures, IB is represented as the smallest relation of which transitive closure is IB. For a given schedule, we denote the jth operation and the entity associated with this operation by $operation(j) \in \{W, R\}$ and $entity(a_j) \in E$, respectively.

The system may provide constructs to allow programmers to specify an order among the operations of different processes. Barrier synchronization primitives [5] and conditional synchronization primitives [4] (e.g., semaphores, continue and delay operations in Concurrent PASCAL, and notify and wait operations in Mesa) are examples for such constructs. We define the relation IB_{inter} to express the order among the operations of different processes specified by the concurrent program. If a and b are operations of processes P_i and P_j

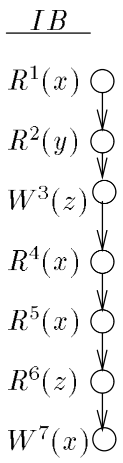

Figure 14: IB originated from the execution of program $K9$, when $x \geq 10$ initially.

```
entities $object$ : character[n];
local $buffer$ : character[n];
local $i, n$ : integer;

read($object, buffer, n$);
for $i = 0$ to $i < n$ do
  $buffer[i] := buffer[i] + 1$;
write($object, buffer, n$);
```

Figure 15: Program $K10$.

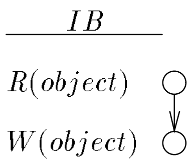

Figure 16: IB originated from the execution of program $K10$.

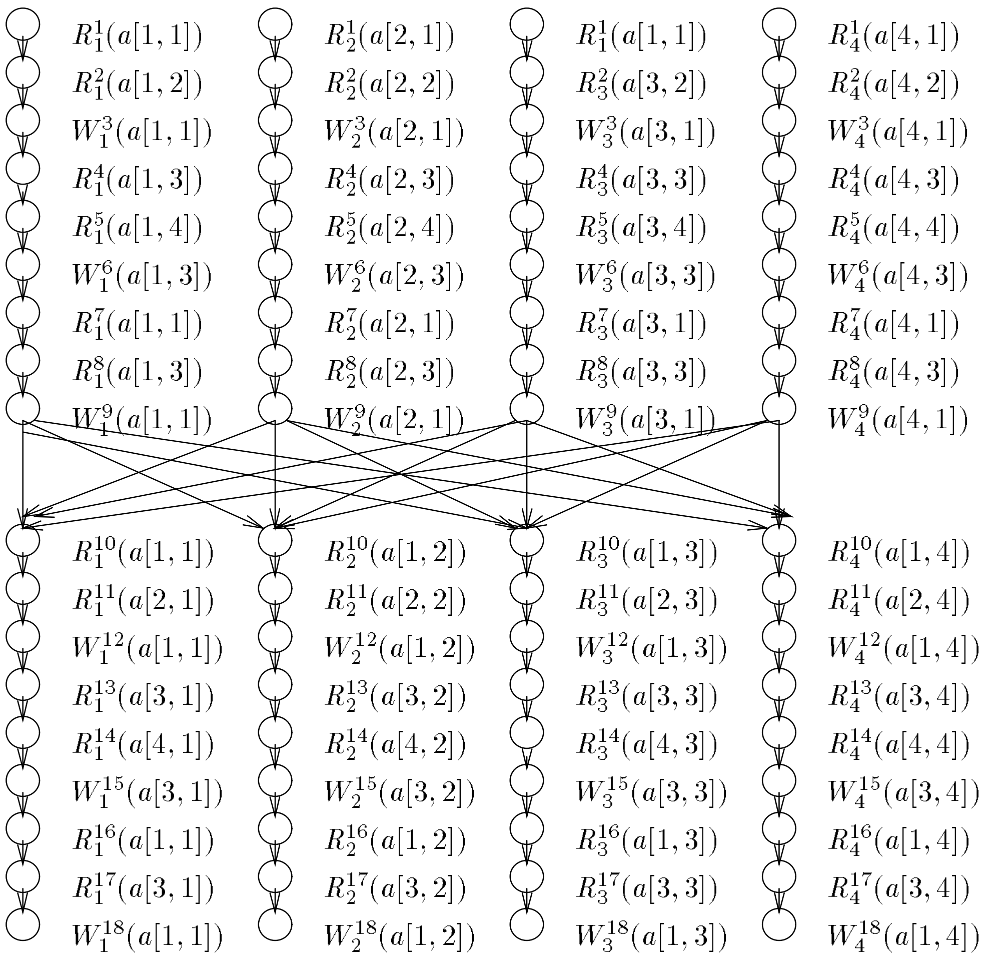

Figure 17: IB for parallel program $K7$

respectively $(i \neq j)$, and the concurrent program specifies that b must be performed after a, then $a\ IB_{inter} b$. For example, consider again Figure 10. Operation $W_1(a[1,1])$ for the third statement of P_1, and operation $R_3(a[1,3])$ for the fourth statement of P_3. The program specifies that $W_1(a[1,1])\ IB_{inter} R_3(a[1,3])$.

We denote the set of all operations of all the processes in a concurrent execution by O, namely $O = \bigcup_{i=1}^{n} O_i$. *Program order* relation IB, denoted by $\xrightarrow{i}$, is defined on O as the transitive closure of $\bigcup_{i=1}^{n} IB_i \cup IB_{inter}$. IB is an irreflexive partial order on O. Note that there can be program orders which cannot be expressed by IB relation. In figures in this paper, we depict IB as the smallest relation of which transitive closure is IB. Figure 17 depicts IB for parallel program in Figure 10.

If the operations are issued in the program order, then IB_i specifies the order in which entity accesses are issued. This is not the case for some pipelined processors that issue operations in different order than the order defined in the code.

We assume that concurrent programs are written correctly for an ideal machine, and that the code generated by the compiler either preserves the order of entity accesses in each program, or if the compiler rearranges the order of entity access in a program, it preserved the interprocess dependencies. Thus, if the entity accesses are executed in the order that is defined in the compiler generated code, the result will be correct. In pipelined processors, branches can affect the pipeline performance [12]. One method for reducing pipeline stalls due to branch delays is predicting branches. According to prediction, the branch is either taken or not taken before the branch condition is calculated. If the prediction is wrong, the state must be restored. For such architectures, we assume that pipeline stalls, if the shared data (entities) must be accessed in a predicted branch until branch condition is computed.

3.3 Performed Before Order

As we pointed out before, the order in which operations are performed can be different from the program order. We define the *performed before* relation PB, denoted by $\xrightarrow{p}$, on O to capture the observable order, in which the accesses are performed, as follows:

1. If a and b are operations of the same process or different processes, and performing b is delayed until a is performed, then $a \xrightarrow{p} b$.

2. If an operation $R(x)$ in a process returns the value written by an operation $W(x)$ in the same process or different processes, then $W(x) \xrightarrow{p} R(x)$.

3. If an operation $W(x)$ in a process overwrites the value read by an operation $R(x)$ in the same process or different processes, then $R(x) \xrightarrow{p} W(x)$.

4. If an operation $W(x)$ in a process overwrites the value written by an operation $W(x)$ in the same process or different processes, then $W(x) \xrightarrow{p} W(x)$.

5. If $a \xrightarrow{p} b$ and $b \xrightarrow{p} c$, then $a \xrightarrow{p} c$.

PB is an irreflexive partial order on O. In figures in this paper, we illustrate PB as the smallest relation of which transitive closure is PB.

The rationale behind the definition of PB is as follows. Item (1) expresses the order imposed due to either architectural assumptions or program specification. For example, some processors do not issue an operation until the previous operations are performed, or some pipelined processes do not issue operations that are dependent on previously issued operations until these operations are performed. Another example is the access ordering in bus-based shared memory multiprocessors. Since there is one FIFO path between a processor and all memory modules, then for any two operations a and b of a process, it is known that if $a \xrightarrow{i} b$, then $a \xrightarrow{p} b$. Yet another example is the fence operation in some shared memory multiprocessors, which allows to delay issuing, and hence, performing of an access until some previous accesses are performed [13]. Item (2) is due to causality. Items (3) and (4) are due to our assumption about the system that data is kept coherent. Item (5) simply expresses the transitivity of PB relation.

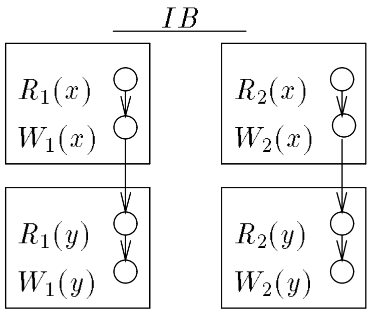

Figure 18: IB order and A relation for program $K4$.

3.4 Atomic Actions

An *atomic action* is a sequence of operations whose execution is guaranteed to yield the same effect as if the operations were executed exclusively. An equivalence relation A_i on O_i for each process P_i specifies the atomic actions. If a and b are operations of the same process, and a and b must performed atomically, then aA_ib. We define the equivalence relation A on the set O of operations of all processes: $A = \bigcup_{i=1}^{n} A_i$.

The following definitions, borrowed from [6], will be used later in the paper. If R is an irreflexive relation and A is an equivalence relation on a set U, then R/A is an irreflexive relation induced by R on the set of equivalence classes U/A. R is defined as

$$R/A = \{(r_1, r_2)|\ r_1 \in U/A\ \wedge\ r_2 \in U/A\ \wedge\ r_1 \neq r_2\ \wedge\ (\exists a \exists b:\ a \in r_1, b \in r_2:\ a\ R\ b)\}$$

Hence, PB/A and IB/A are the irreflexive relations induced by PB and IB on the set of equivalence classes O/A. We assume that IB/A is a partial order, and IB/A specifies the program order among the atomic actions in a concurrent execution. In figures that depict IB, we illustrate the operations in an atomic action within a box. Consider again concurrent program $K4$ of Figure 4. The IB order for program $K4$ is depicted in Figure 18, where $A_1 = \{(R_1(x), W_1(x)), (R_1(y), W_1(y))\}$ and $A_2 = \{(R_2(x), W_2(x)), (R_2(y), W_2(y))\}$.

3.5 Data Dependence

We say that a write operation $W_i^k(x)$ is *dependent on an entity* y (where y can be equal to x), if the value of entity y is used to compute the value of x. We say that $W_i^k(x)$ is *dependent on read operation* $R_i^l(y)$, if it is dependent on y and $R_i^l(y)$ is is the first read on y before $W_i^k(x)$ in IB_i. Note that we have defined dependency within a process.

To illustrate, consider program $K9$ in Figure 13 and the corresponding schedule in Figure 14. $W^3(z)$ is dependent on entities x and y and on operations $R^1(x)$ and $R^2(y)$, whereas $W^7(x)$ is dependent on entities x and z and on operations $R^5(x)$ and $R^6(z)$. For program $K10$ in Figure 15 and its schedule in Figure 16, $W(object)$ is dependent on entity *object* and on operation $R(object)$.

We assume that the system stalls before issuing a write operation until all reads that write is dependent are performed. Hence, for any write operation a that is dependent on a read operation b, $b \xrightarrow{p} a$.

3.6 Interpretation

We borrow the notion of interpretation from [7]. The interpretation of a schedule (IB_i) is specified by the program of process P_i from which the schedule is originated. An interpretation of a schedule is a pair $I_i = (D, F)$, where $D = \{D_x, D_y, ...\}$ is a set of *domains*, one for each entity in $\mathbf{E}$; each domain is a set of values for the corresponding entity. F is a set of *functions*, one for each write operation in IB_i and is defined as:

$$F = \{f_j | \ j \text{ is a step of the schedule and } operation(j) = W\}$$

For each such step j, f_j is a mapping

$$f_j : \prod_{x \in B(j)} D_x \longrightarrow D_{entity(j)},$$

where

$$B(j) = \{x \in E | \ W^j(entity(j)) \text{ is dependent on } x\}.$$

We illustrate this concept by an example. Consider again program $K9$ in Figure 13. Figure 14 depicts the schedule generated by executing program $K9$, when initially $x \geq 10$. The interpretation $I_i = (D, F)$ of the schedule corresponding to program $K9$ is the following. The domains D_x, D_y, D_z for the entities x, y, z are the set of integers, and the functions corresponding to the write operations are $f_3(x, y) = x + y$ and $f_7(z, x) = z - x$.

3.7 Concurrent Execution, Correct Execution and Execution Model

Two relations R_1 and R_2 are said to be *consistent,* if $R_1 \cup R_2$ can be extended to a total ordering. A relation can be extended to a total ordering if and only if its transitive closure is irreflexive. Thus, R_1 is consistent with R_2 if and only if $R_1 \cup R_2$ is acyclic.

A *concurrent execution s* is represented by a tuple $< C, PB >$, where C is a tuple $< O, I, A, IB >$ that specifies the correct execution. We refer to C as the *correctness criterion* or the *specification of the correct execution.* O is the set of entity accesses of all processes in the concurrent execution. A is the equivalence relation on O that defines the atomic actions. IB and PB are the program and performing orders on the entity accesses O, respectively. I is the set consisting of the union of all interpretations of all schedules in the concurrent execution.

We have assumed that programs are written correctly for the ideal system. Hence, we know that an execution in which operations are performed in the program order, and in which atomic actions are executed exclusively is correct. Formally, we can define the correct execution as follows.

Definition 3.1 *For a given specification $C =< O, I, A, IB >$, an execution $s =< C, PB >$ is* correct *if the result of s is equal to the result of any execution in the set of executions X, where*

$$\begin{array}{lll} X = & \{< C, PB_c > | & IB \subseteq PB_c \quad \wedge \\ & & PB_c \textit{ is consistent with } IB \quad \wedge \\ & & IB/A \subseteq PB_c/A \quad \wedge \\ & & PB_c/A \textit{ is consistent with } IB/A \\ & \}. & \end{array}$$

An *execution model* describes an execution order on operations of processes, such that the result of an execution, which adheres this model, is the same as if the operations were executed in this order. An execution model can be specified by a type of atomic constraints and a type program order constraints. Execution $s =< C, PB >$, which adheres the model, is correct, if the constraints of the model match the ones in the specification C of the correct execution. Let A_e be the equality relation. For example, sequential consistency is an execution model in which atomic constraints are specified by A_e and program order constraints are specified by $\bigcup_{i=1}^{n} IB_i$, where IB_i is the program order of process P_i.

Definition 3.2 *For a given specification $C =< O, I, A, IB >$, an execution $s =< C, PB >$ is* sequentially consistent, *if the result of s is equal to the result of a sequential execution defined as*

$$\begin{array}{lll} s_c = & \{< C_c, PB_c > \mid & C_c =< O, I, \bigcup_i IB_i, A_e > \quad \wedge \\ & & \bigcup_i IB_i \subseteq PB_c \quad \wedge \\ & & PB_c \textit{ is consistent with } \bigcup_i IB_i \\ & \}. & \end{array}$$

Let A_s be the equivalence relation defined as follows: $a\ A_s\ b$, if and only if a and b are operations of the same process.

Definition 3.3 *For a given specification $C =< O, I, A, IB >$, an execution $s =< C, PB >$ is* serializable, *if the result of s is equal to the result of a serial execution defined as*

$$\begin{array}{lll} s_s = & \{< C_s, PB_s > \mid & C_s =< O, I, \bigcup_i IB_i, A_s > \quad \wedge \\ & & \bigcup_i IB_i \subseteq PB_s \quad \wedge \\ & & PB_s \textit{ is consistent with } \bigcup_i IB_i \quad \wedge \\ & & (\bigcup_i IB_i)\ /A_s \subseteq PB_s\ /A_s \quad \wedge \\ & & PB_s\ /A_s \textit{ is consistent with } (\bigcup_i IB_i)\ /A_s \\ & \}. & \end{array}$$

4 Correctness, Sequential Consistency, Serializability

Our theory encompasses sequentially consistent executions of programs. Sequential consistency is the typical execution model for multiprocessor systems. The notion of sequential consistency was defined in [1] as follows: *" The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* In other words, an execution is sequentially consistent, if its result is equivalent to a sequential execution. For a given program, if A is the equality relation and $IB_{inter} = \emptyset$, then a sequentially consistent execution is correct.

For applications in which A and IB_{inter} are specified differently, sequential consistency is not sufficient to ensure correctness. These application domains include those programs which are specified as a sequence of atomic segments, each of which is a sequence of indivisible operations (e.g., programs in Figures 4 and 5) and those programs in which there is a specific order specified among the operations of different processes (eg., programs in Figures 10 and 11). A sequentially consistent execution of such programs may yield incorrect results.

Our theory encompasses serializable executions of programs. Serializability is the typical execution model that is used in database systems. Serializability is defined as follows. *The result of any execution is the same as if the processes (transactions) were executed in some serial order.* In other words, an execution is serializable, if its result is equivalent to a serial execution. If A is equal to relation A_s defined in Section 3.7, and $IB_{inter} = \emptyset$, then a serializable execution is correct.

For applications in which IB_{inter} is not an empty set, serializability is not sufficient to ensure correctness. These application domains include those programs in which there is a specific order specified among the operations of different processes (eg., programs in Figures 10 and 11). A serializable execution of such programs may yield incorrect results. Furthermore, for applications, in which $IB_{inter} = \emptyset$, but A is defined differently, serializability degrades the performance unnecessarily. These application domains include those programs which are specified as a sequence of atomic segments (e.g., program in Figures 4).

In addition to sequential consistency and serializability, one can define other execution models with different atomic constraints and program order constraints. Our theory allows us to deal with any execution model, in which atomic constraints can be expressed as an equivalence relation and program order constraints as a partial order. Our goal is to derive algorithms to test whether an execution is correct or meets a given execution model. To this end, we develop different notions of equivalence, and propose extensions to some of the concepts in serializability theory. We make three extensions to read-write model of transactions. These extension also require that changes be made to the various testing algorithms for serializability [2, 7]. Since serializability is a special case of our general correctness criterion, the testing algorithms that will be presented in Section 7 also cover the necessary changes to the testing algorithms for serializability with the following extensions.

1. We remove the restriction in the transaction model that each transaction reads and writes an entity at most once, and if a transaction reads and writes an entity x, $W(x)$ follows $R(x)$. Hence, the schedule of a process can have more than one read and write in any order.

2. We remove the restriction in the transaction model that if a transaction issues operation a before operation b, then a is executed before b.

3. We assume a write is dependent only on a subset of previous reads in the schedule. The subset of reads is defined by the dependencies. In serializability, a write is assumed to be dependent on all the previous reads in the schedule [7]. Hence, we define the equivalence classes for a set of interpretations that result in the same dependencies, whereas in serializability, the equivalence classes are defined for all possible interpretations of an schedule. The latter definition is more conservative in the sense that the set of correct executions accepted under this definition is a subset of correct executions accepted under our definition.

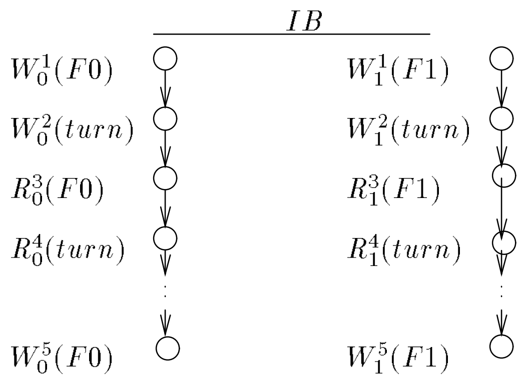

Figure 19: IB originated from an execution of $K2$.

```
entities x, y : integer;

x := x + 10;
y := y + 10;
```

Figure 20: Program $K11$.

The first extension allows us to capture typical concurrent programs in which processes interact through reading and writing shared entities. This is in contrast to the transaction model, in which the goal is to execute each transaction in isolation. For example, consider Peterson's solution to two-process critical section problem as shown in Figure 2. In Figure 19, the schedules originated from an execution of program $K2$ are depicted. In this execution, $F0$ and $F1$ are *false* initially, and process P_0 reads $F0$ before process P_1 starts executing. Entities $F0$ and $F1$ are written twice in IB_0 and IB_1, respectively, and entity *turn* is read after it is written in both schedules.

The second extension is introduced to allow the pipelining of operations, whereas the third extension is introduced to increase pipelining. For example, consider program $K11$ in Figure 20 and its IB in Figure 21. In our model, $W(y)$ is dependent only on $R(y)$, whereas in serializability theory, $W(y)$ is assumed to be dependent on both $R(y)$ and $R(x)$. If pipelining is allowed, $W(y)$ can be issued or performed before $R(x)$ is issued or performed according to our theory, whereas $W(y)$ can only be issued after both $R(x)$ and $R(y)$ are performed according to the serializability theory.

In the remainder of this paper, we only refer to an atomic action which contains more than one operation as an atomic action. $O_i/A_i - O_i/A_e$ is the set of atomic actions in process P_i. We denote this set by AA_i. Hence, $AA_i = O_i/A_i - O_i/A_e = \{O_i^1, O_i^2, ...\}$, where O_i^j is the set of operations in atomic action j in process P_i. The set of all atomic actions is denoted by AA, where $AA = \bigcup_i AA_i$.

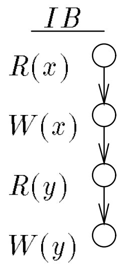

Figure 21: IB originated from execution of $K11$.

5 Classes of Correct Executions

In this section, we present three different notions of equivalence to categorize correct executions and executions that meet a given execution model into classes. The containment relation between these classes is in terms of the restriction placed on the access order.

5.1 View Correctness

We redefine the notion of view equivalence used in serializability theory [2] to capture the case where a process can issue several read and write operations on the same entity in any order, and operations can be performed in an order different from the order in which they are issued.

Definition 5.1 *Two executions $s_1 =< C_1, PB_1 >$ and $s_2 =< C_2, PB_2 >$ are view equivalent, if $O_1 = O_2$, $I_1 = I_2$, and*

1. *for each entity x, if $R_i^j(x)$ returns the initial value of x in execution s_1, then $R_i^j(x)$ must return the initial value of x in execution s_2, and*
2. *for each entity x, if $R_i^j(x)$ returns the value written by $W_k^l(x)$ in execution s_1, then $R_i^j(x)$ must return the value written by $W_k^l(x)$ in execution s_2, and*
3. *for each entity x, if $W_i^j(x)$ writes the final value of x in execution s_1, then $W_i^j(x)$ must write the final value of x in execution s_2, and*
4. *for each entity x, if $W_i^j(x)$ is dependent on entity y and $R_i^k(y)$ is the first read performed on y before $W_i^j(x)$ in execution s_1, then $R_i^k(y)$ must be the first read performed on y before $W_i^j(x)$ in execution s_2.*

Definition 5.2 *For a given C, let X be the set of executions as defined in Section 3.7. Execution $s =< C, PB >$ is* view correct, *if it is view equivalent to an execution in X.*

Definition 5.3 *An execution is* view consistent, *if it is view equivalent to a sequential execution.*

View consistency should not be confused with *view serializability*. An execution is view serializable, if it is view equivalent to a serial execution.

5.2 B Correctness

We will prove in Section 7.1 that testing for view correctness is an NP-complete problem. Therefore, we must search for other classes of correct executions that restrict the access order more than view correct executions. For this purpose, we define a new equivalence notion— B equivalence. In Section 7.2, we will present a polynomial time testing algorithm for B correctness.

Definition 5.4 *Two executions $s_1 =< C_1, PB_1 >$ and $s_2 =< C_2, PB_2 >$ are B equivalent, if*

1. *s_1 and s_2 are view equivalent, and*
2. *for each entity x, if $W_i^j(x)$ is performed before $W_k^l(x)$, and if $R_p^q(x)$ returns the value written by $W_k^l(x)$ in execution s_1, then $W_i^j(x)$ must be performed before $W_k^l(x)$ in s_2, and*
3. *for each entity x, if $W_i^j(x)$ is performed after $R_p^q(x)$ in execution s_1, then $W_i^j(x)$ must be performed after $R_p^q(x)$ in s_2.*

Definition 5.5 *For a given C, let X be the set of executions as defined in Section 3.7. Execution $s =< C, PB >$ is B correct, if s is B equivalent to an execution in X.*

Definition 5.6 *An execution s is B consistent, if s is B equivalent to a sequential execution.*

Definition 5.7 *An execution s is B serializable, if s is B equivalent to a serial execution.*

Theorem 5.1 *If an execution s is B correct, then s is view correct.* $\square$

The converse of Theorem 5.1 is not true. Similarly, if an execution s is B consistent, then s is view consistent, and if an execution s is B serializable, then s is view serializable, and the converses of these statements are not correct.

5.3 Conflict Correctness

Although B correctness yields a polynomial time testing algorithm, for completeness, we introduce another correctness class—conflict correctness, which also yields a polynomial time testing algorithm, but restricts the access order more than B correctness. Conflict equivalence is widely used in serializability theory [2] and in optimization techniques for parallel programs [6].

Two operations are said to be *conflicting*, if they access the same entity and at least one of them is a write. We extend the definition of conflict equivalence defined for databases [2].

Definition 5.8 *Two executions $s_1 =< C_1, PB_1 >$ and $s_2 =< C_2, PB_2 >$ are conflict equivalent, if*

1. *s_1 and s_2 are B equivalent, and*

2. *for each entity x, if $W_k^l(x)$ overwrites the value returned by $R_i^j(x)$ in execution s_1, then $W_k^l(x)$ must overwrite the value returned by $R_i^j(x)$ in execution s_2.*

3. *for each entity x, if $W_k^l(x)$ overwrites the value written by $W_i^j(x)$ in execution s_1, then $W_k^l(x)$ must overwrite the value written by $W_i^j(x)$ in execution s_2.*

Definition 5.9 *For a given C, let X be the set of executions as defined in Section 3.7. Execution $s =< C, PB >$ is* conflict correct, *if it is conflict equivalent to an execution in X.*

Definition 5.10 *An execution s is* conflict consistent, *if s is conflict equivalent to a sequential execution.*

Conflict consistency should not be confused with *conflict serializability.* An execution is conflict serializable, if it is conflict equivalent to a serial execution.

Theorem 5.2 *If an execution is conflict correct, then it is B correct.* □

The converse of Theorem 5.2 is not true. Similarly, if an execution s is conflict consistent, then s is B consistent, and if an execution s is conflict serializable, then s is B serializable, and converses of these statements are not correct.

6 Hierarchical Graphs and Polygraphs

In order to reduce the complexity of testing for correctness, we define the concepts of *hierarchical graph* and *hierarchical polygraphs.* Informally, a hierarchical graph is a graph in which some nodes are themselves graphs. A hierarchical polygraph is a polygraph (polygraphs are defined in [7]), in which some nodes are themselves polygraphs. Each hierarchical graph represent a graph, which we refer as the *underlying graph.*

Definition 6.1 *A hierarchical graph (or* higraph*) is a tuple $\mathcal{G} = (V_1, V_0, E)$. V_1 is the set of* supernodes. *Each supernode is a graph. Hence,*

$$V_1 = \{(V_{1i}, E_{1i})\}$$

V_0 is the set of nodes, and E is the set of edges defined by a binary relation on $V_1 \cup V_0$.

In figures, we depict a higraph $\mathcal{G} = (V_1, V_0, E)$ with a set of graphs. The set includes graph $G' = (V_1 \cup V_0, E)$, in which nodes corresponding to supernodes are drawn in black, and a graph per supernode, which is drawn in a circle. Figure 22 illustrates a higraph. In this higraph, the set of supernodes is $V_1 = \{sn_1, sn_2\}$, where $sn_1 = (V_{11}, E_{11})$, $V_{11} = \{n_1, n_2\}$, $E_{11} = \emptyset$. $sn_2 = (V_{12}, E_{12})$, $V_{12} = \{n_3, n_4, n_5, n_6\}$, and $E_{12} = (n_3, n_5), (n_4, n_5), (n_5, n_6)\}$. The set of nodes V_0 is empty. The set of edges is $E = \{(sn_1, sn_2)\}$.

To simplify the definition of the underlying graph of a higraph, we define the function *parent* for a higraph from the set $\bigcup_i V_{1i} \cup V_0$ to the set $V_1 \cup V_0$ as follows:

$$parent(n) = \begin{cases} sn & \text{if } (\exists sn : \ sn \in V_1 : \ n \in sn) \\ n & \text{otherwise} \end{cases}$$

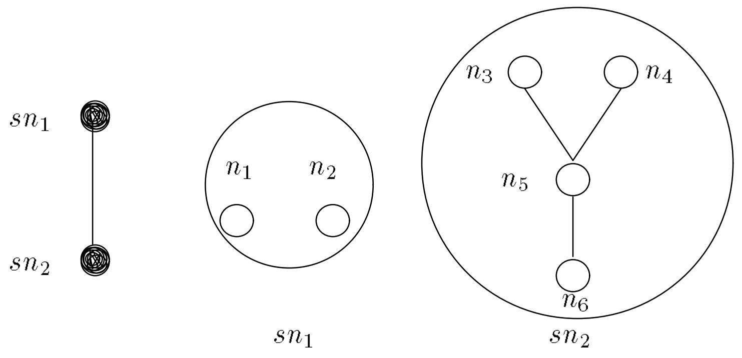

Figure 22: Higraph $\mathcal{G}_1$.

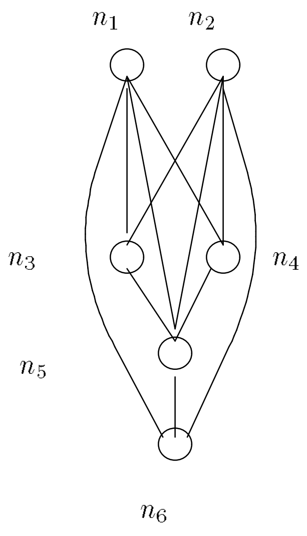

Figure 23: G is the underlying graph of higraph $\mathcal{G}_1$.

Definition 6.2 *A higraph $\mathcal{G} = (V_1, V_0, E)$ represents an underlying graph $G = (V, E')$, where*

$$V = \bigcup_i V_{1i} \ \cup V_0,$$

$$E' = \bigcup_i E_{1i} \ \cup E'',$$

$$E'' = \{(n_i, n_j)| \ n_i \in V \wedge n_j \in V \wedge (parent(n_i) \neq parent(n_j)) \wedge ((parent(n_i), parent(n_j)) \in E)\}.$$

Figure 23 illustrates the underlying graph of higraph $\mathcal{G}_1$ depicted in Figure 22. For higraph $\mathcal{G}_1$, $parent(n_1) = parent(n_2) = sn_1$ and $parent(n_3) = parent(n_4) = parent(n_5) = parent(n_6) = sn_2$. For the underlying graph G, $E'' = \{(n_1, n_3), (n_1, n_4), (n_1, n_5), (n_1, n_6), (n_2, n_3), (n_2, n_4), (n_2, n_5), (n_2, n_6)\}$.

Definition 6.3 *A hierarchical polygraph (or* hipolygraph*) is a tuple $\mathcal{P} = (V_1, V_0, E, C)$. V_1 is the set of* supernodes. *Each supernode is a polygraph. Hence,*

$$V_1 = \{(V_{1i}, E_{1i}, C_{1i})\}$$

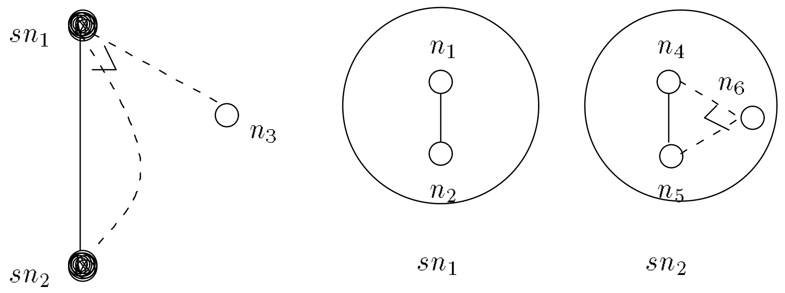

Figure 24: A hipolygraph $\mathcal{P}_1$.

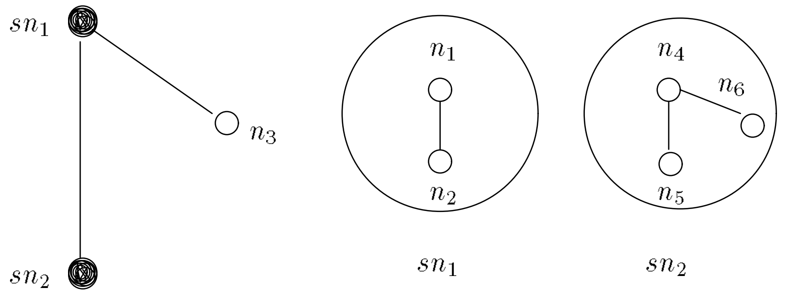

Figure 25: A higraph that is compatible with hipolygraph $\mathcal{P}_1$.

V_0 is the set of nodes, and E is the set of edges defined by a binary relation on $V_1 \cup V_0$. C is the set of choices defined on $V_1 \cup V_0$.

In figures, we depict a hipolygraph $\mathcal{P} = (V_1, V_0, E, C)$ with a set of polygraphs. The set includes polygraph $P' = (V_1 \cup V_0, E, C)$, in which nodes corresponding to supernodes are drawn in black, and a polygraph per supernode, which is drawn in a circle. Figure 24 illustrates a hipolygraph. In this hipolygraph, the set of supernodes is $V_1 = \{sn_1, sn_2\}$, where $sn_1 = (V_{11}, E_{11}, C_{11})$, $V_{11} = \{n_1, n_2\}$, $E_{11} = \{(n_1, n_2)\}$, $C_{11} = \emptyset$, $sn_2 = (V_{12}, E_{12}, C_{12})$, $V_{12} = \{n_4, n_5, n_6\}$, $E_{12} = \{(n_4, n_5)\}$, and $C_{12} = \{(n_4, n_6, n_5)\}$. The set of nodes is $V_0 = \{n_3\}$, the set of edges $E = \{(sn_1, sn_2)\}$, and the set of choices $C = \{(n_3, sn_1, sn_2)\}$. The function *parent* is defined the same way for a hipolygraph. For hipolygraph $\mathcal{P}_1$, $parent(n_1) = parent(n_2) = sn_1$, $parent(n_3) = n_3$, and $parent(n_4) = parent(n_5) = parent(n_6) = sn_2$.

Definition 6.4 *Higraph $\mathcal{G} = (V_1, V_0, E)$ is said to be compatible with hipolygraph $\mathcal{P} = (V'_1, V_0, E', C')$, if*

1. *$E' \subseteq E$, and for each choice (c_1, c_2, c_3) in C', either edge (c_1, c_2) or edge (c_2, c_3) is in E, and*

2. *$E'_{1i} \subseteq E_{1i}$, and for each choice (c_1, c_2, c_3) in C'_{1i}, either edge (c_1, c_2) or edge (c_2, c_3) is in E_{1i}.*

Figure 25 illustrates a higraph that is compatible with hipolygraph $\mathcal{P}_1$ depicted in Figure 24.

Definition 6.5

1. *A supernode is said to be acyclic, if the graph that the supernode contains is acyclic.*

2. *A higraph is said to be acyclic, if the underlying graph is acyclic.*

3. *A hipolygraph is said to be acyclic, if there is a compatible higraph which is acyclic.*

4. *A supernode is said to be a total order, if the graph that the supernode contains is a total order.*

5. *A higraph is said to be a total order, if the underlying graph is a total order.*

Theorem 6.1 *A higraph $\mathcal{G} = (V_1, V_0, E)$ is acyclic, if and only if each supernode in V_1 is acyclic, and graph $G' = (V_1 \cup V_0, E)$ is acyclic.* $\square$

7 Testing Algorithms

In this section, we develop algorithms for testing view correctness, B correctness and conflict correctness. These algorithms can also be used to test whether an execution meets a given execution model by replacing the atomic constraints and program order of the specification of correct execution by the ones of the execution model.

7.1 View Correctness

In order to test whether an execution s is view correct, we define a directed hipolygraph $\mathcal{P}(s)$. An *augmented execution* $\hat{s}$ of an execution s contains two new processes P_b and P_f, besides those in s. P_b consists of only write steps, one for each entity read or written in s. P_f consists of only read operations one for each entity read or written in s. Execution $\hat{s}$ starts with P_b and ends with P_f. Given an execution s, hipolygraph $\mathcal{P}(s) = (V_1, V_0, E, C)$ is constructed as follows. In $\mathcal{P}(s)$, there are two nodes that represent P_b and P_f respectively, and one node for each operation in s which is not in an atomic action. Hence, $V_0 = \{P_b, P_f\} \cup O - \bigcup_{S \in AA} S$. Each supernode corresponds to an atomic action. The set of supernodes is $V_1 = \{(O_1^1, E_1^1, C_1^1), (O_1^2, E_1^2, C_1^2), ..., (O_2^1, E_2^1, C_2^1), ..., \}$, where O_i^j is the operations in atomic action j of process P_i. There are six types of directed edges in E_i^j.

1. For each pair of operations a and b in atomic action j in process P_i, if a is immediately before b in program order IB, then the arc (a, b) is added to E_i^j.

2. For each entity x, if $R_i^k(x)$ and $W_i^l(x)$ are operations in atomic action j in process P_i, and $R_i^k(x)$ returns the value written by $W_i^l(x)$, then $(W_i^l(x), R_i^k(x))$ is added to E_i^j.

3. For each entity x, if $R_i^k(x)$ and $W_i^m(x)$, are operations in atomic action j in process P_i, $R_i^k(x)$ returns the value written by $W_a^l(x)$ that is not in the same atomic action, then edge $(R_i^k(x), W_i^m(x))$ is added to E_i^j.

4. For each entity x, if $W_i^l(x)$ and $W_i^m(x)$, are operations in atomic action j in process P_i, $R_a^k(x)$ that is not in the same atomic action returns the value written by $W_i^l(x)$, then edge $(W_i^m(x), W_i^l(x))$ is added to E_i^j.

5. For each entity x, if there is $R_i^k(x)$ in atomic action j in process P_i that returns the initial value of x, and there is a $W_i^l(x)$ in the same atomic action, then edge $((R_i^k(x), W_i^l(x))$ is added to E_i^j.

6. For each entity x, if there is $W_i^k(x)$ in atomic action j in process P_i that writes the final value of x, and there is a $W_i^l(x)$ in the same atomic action, then edge $(W_i^l(x)), W_i^k(x)))$ is added to E_i^j.

For each entity x, if $R_i^k(x), W_i^l(x)$, and $W_i^m(x)$, are operations in atomic action j in process P_i, and $R_i^k(x)$ returns the value written by $W_i^l(x)$, then choice $(R_i^k(x), W_i^m(x), W_i^l(x))$ is added to C_i^j.

There are six types of directed edges in E.

1. For each operation a in any process in s, the arc $(P_b, parent(a))$ is added to E.

2. For each operation a in in s, the arc $(parent(a), P_f)$ is added to E.

3. For each pair of operations a and b in any processes in s, if $parent(a) \neq parent(b)$, and a is before b in program order IB, and there is no other operation c in any process in s, such that $parent(a) \neq parent(c) \neq parent(b)$, and a is before c and c is before b in IB, then the arc $(parent(a), parent(b))$ is added to E.

4. For each entity x, if $R_i^j(x)$ in any process in s returns the value written by $W_k^l(x)$ in any process in s, and $parent(R_i^j(x)) \neq parent(W_k^l(x))$, then edge $(parent(W_k^l(x), parent(R_i^j(x))$ is added to E.

5. For each entity x, if in any process in s, there is a read operation $R_i^j(x)$ that returns the initial value of x, and there is a write operation $W_k^l(x)$ in any process in s, and $parent(R_i^j(x)) \neq parent(W_k^l(x))$, then edge $(parent(R_i^j(x)), parent(W_k^l(x)))$ is added to E.

6. For each entity x, if in any process in s, there is a write operation $W_i^j(x)$ that writes the final value of x, and there is another write operation $W_k^l(x)$ in any process in s, and $parent(W_i^j(x)) \neq parent(W_k^l(x))$, then edge $(parent(W_k^l(x)), parent(W_i^j(x)))$ is added to E.

The set of directed choices C is constructed as follows: For each entity x and operations $R_i^j(x)$, $W_k^l(x)$, and $W_c^a(x)$ in any processes in s, such that $R_i^j(x)$ returns the value written by $W_k^l(x)$, then

$$(parent(R_i^j(x)), parent(W_c^a(x)), parent(W_k^l(x)))$$

is added to the set of choices C, if

1. $parent(R_i^j(x)) = parent(W_k^l(x)) \neq parent(W_c^a(x))$, or

2. $parent(R_i^j(x)) \neq parent(W_c^a(x)) \neq parent(W_k^l(x)) \neq parent(R_i^j(x))$.

Theorem 7.1 *An execution s is view correct if and only if $\mathcal{P}(s)$ is acyclic.* □

Theorem 7.2 *The problem of deciding whether an execution is view correct is NP-complete.* □

P_1	P_2
entities $x1, x2, y1$: **integer**;	**entities** x, y : **integer**;
local t : **integer**;	**local** t : **integer**;
$t := y1$;	$t := y2$;
$x1 := x1 - t$;	$x1 := x1 + t$;
barrier synchronization;	barrier syncronization;
$x2 := x1 - x2$;	$x2 := x1 + x2$;

Figure 26: Program $K12$ that is written with the assumption that each increment statement is atomic.

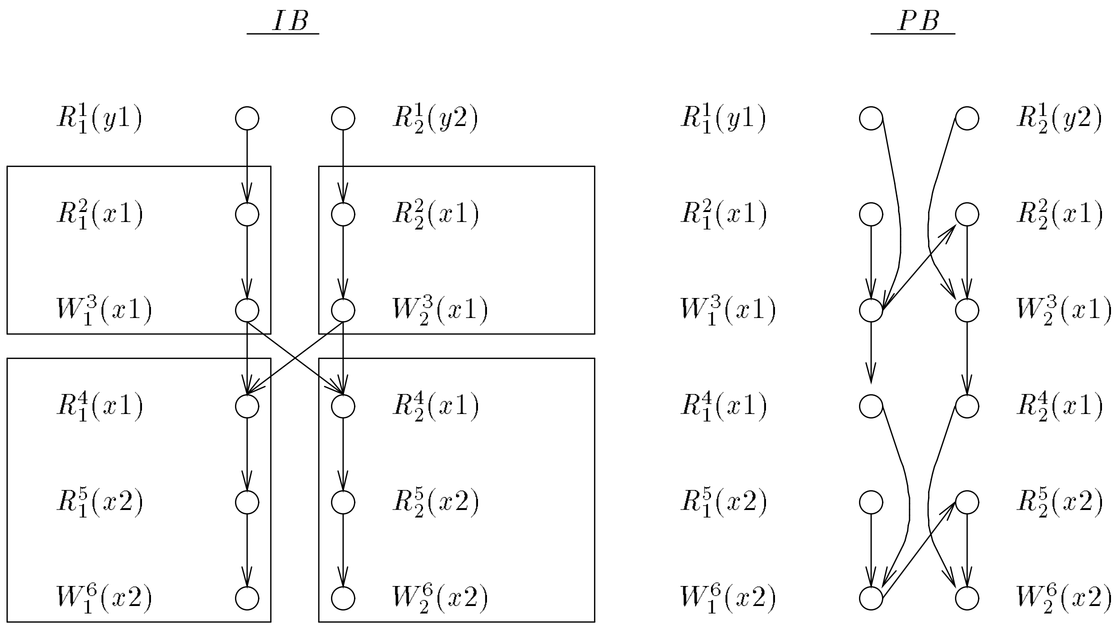

Figure 27: s_{12}: An execution of program $K12$.

In Figure 26, we define a parallel program $K12$ in which barrier synchronization is used to order operations of two processes, and which is written with the assumption that increment statements are atomic. Figure 27 illustrates an execution s_{12} of program $K12$, and in Figure 28, we show the hipolygraph $\mathcal{P}(s_{12})$ corresponding to execution s_{12}. The hipolygraph is cyclic, therefore the execution is not view correct. Figure 29 illustrates execution s_{13}, which is view correct. Note that if an execution $s =< C, PB >$ does not contain any atomic actions, testing hipolygraph $\mathcal{P}(s) = (V_1, V_0, E, C)$ becomes a regular polygraph $P(s) = (V_0, E, C)$. In this case, execution s is both view correct and view consistent.

7.2 B Correctness

We define the directed higraph $\mathcal{H}(s)$ to test whether execution s is B correct. Given an execution s, higraph $\mathcal{H}(s) = (V_1, V_0, E)$, where $V_1 = \{(O_1^1, E_1^1), (O_1^2, E_1^2), ..., (O_2^1, E_2^1), ..., \}$, is constructed from hipolygraph

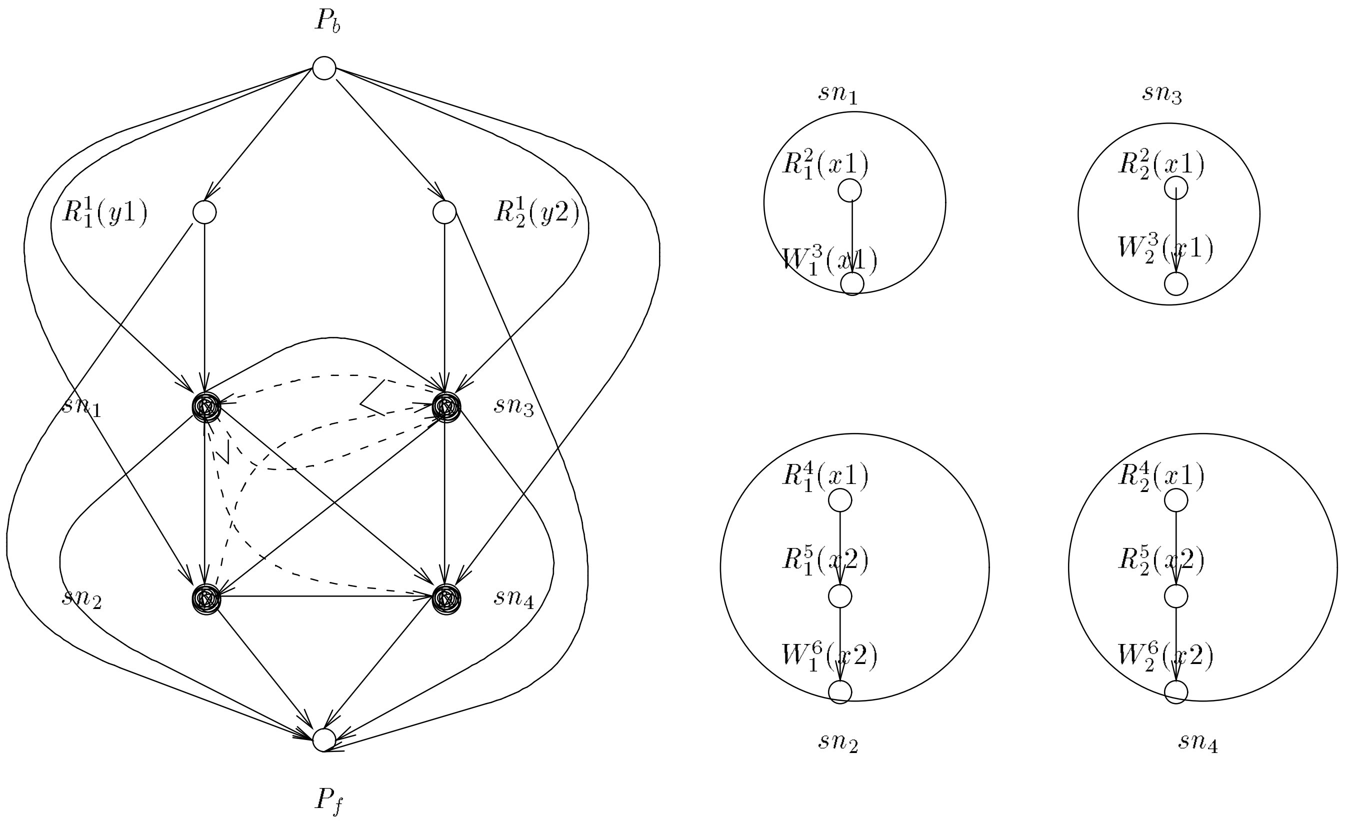

Figure 28: $\mathcal{P}(s_{12})$

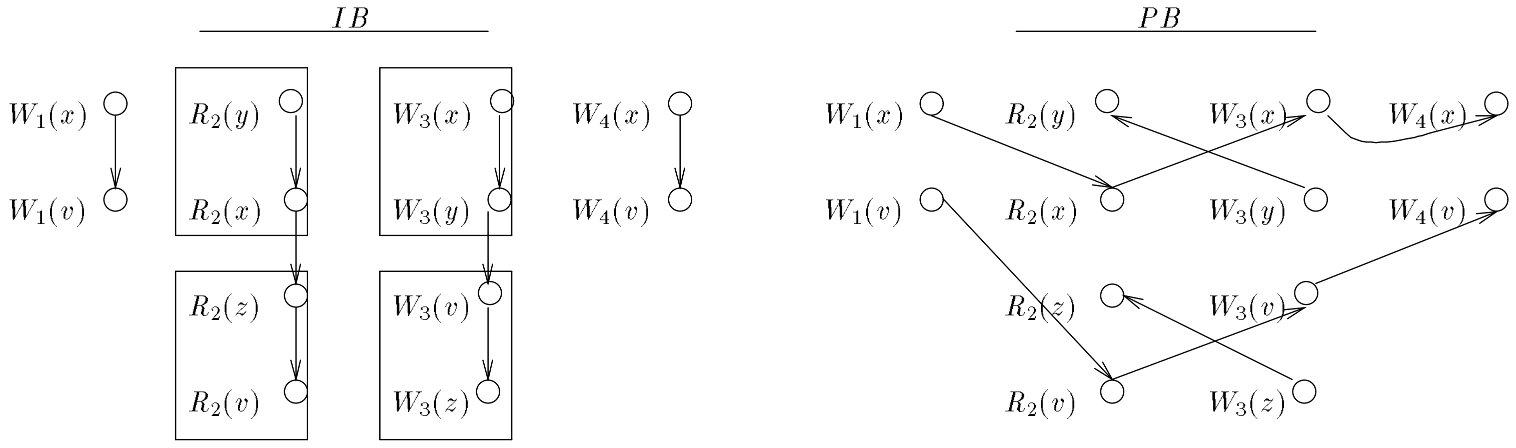

Figure 29: Execution s_{13}.

$\mathcal{P}(s) = (V'_1, V_0, E', C)$, where $V'_1 = \{(O_1^1, E'^1_1, C_1^1), (O_1^2, E'^2_1, C_1^2), ..., (O_2^1, E'^1_2, C_2^1), ..., \}$, as follows.

1. For each supernode, $E'^j_i \subseteq E_i^j$.
2. For each entity x and operations $R_i^k(x), W_i^l(x)$ and $W_i^m(x)$ in atomic action j in process P_i, if $R_i^k(x)$ returns the value written by $W_i^l(x)$,
   (a) arc $(R_i^k(x), W_i^m(x))$ is added to E_i^j, if $W_i^m(x)$ is performed after $R_i^k(x)$,
   (b) arc $(W_i^m(x), W_i^l(x))$ is added to E_i^j, if $W_i^m(x)$ is performed before $W_i^l(x)$.
3. $E' \subseteq E$.
4. For each entity x and operations $R_i^j(x)$, $W_k^l(x)$ and $W_c^a(x)$ in s, such that $R_i^j(x)$ returns the value written by $W_k^l(x)$,
   (a) arc $(parent(R_i^j(x)), parent(W_c^a(x)))$ is added to E, if $W_c^a(x)$ is performed after $R_i^j(x)$, and either $parent(R_i^j(x)) = parent(W_k^l(x)) \neq parent(W_c^a(x))$ or $parent(R_i^j(x)) \neq parent(W_c^a(x)) \neq parent(W_k^l(x)) \neq parent(R_i^j(x))$.
   (b) arc $(parent(W_c^a(x)), parent(W_k^l(x)))$ is added to E, if $W_k^l(x)$ is performed after $W_c^a(x)$, and either $parent(R_i^j(x)) = parent(W_k^l(x)) \neq parent(W_c^a(x))$ or $parent(R_i^j(x)) \neq parent(W_c^a(x)) \neq parent(W_k^l(x)) \neq parent(R_i^j(x))$.

Theorem 7.3 *An execution s is B correct, if and only if $\mathcal{H}(s)$ is acyclic.* □

Theorem 7.4 *We can test whether an execution is B correct in $O(n^2)$ time, where n is the total number operations in all processes.* □

In Figure 30, we show the higraph $\mathcal{H}(s_{13})$ corresponding to execution s_{13} in Figure 29. The higraph is cyclic, and therefore the execution is not B correct. Figure 31 illustrates execution s_{14}, which is B correct. Note that if an execution $s =< C, PB >$ does not contain any atomic actions, testing higraph $\mathcal{H}(s) = (V_1, V_0, E)$ becomes a regular graph $H(s) = (V_0, E)$. In this case, execution s is both B correct and B consistent.

7.3 Conflict Correctness

We define the directed higraph $\mathcal{G}(s) = (V_1, V_0, E)$ to test whether s is conflict correct. The set of vertices V_0 is the set of all operations of all processes in s, which are not in an atomic action. Hence, $V_0 = O - \bigcup_{S \in AA} S$. Each supernode corresponds to an atomic action, hence, the set of supernodes is $V_1 = \{(O_1^1, E_1^1), (O_1^2, E_1^2), ..., (O_2^1, E_2^1), ...\}$. There are four types of directed edges in E_i^j.

1. For each pair of operations a and b in atomic action j in process P_i, if a is immediately before b in the program order, then the arc (a, b) is added to E_i^j.

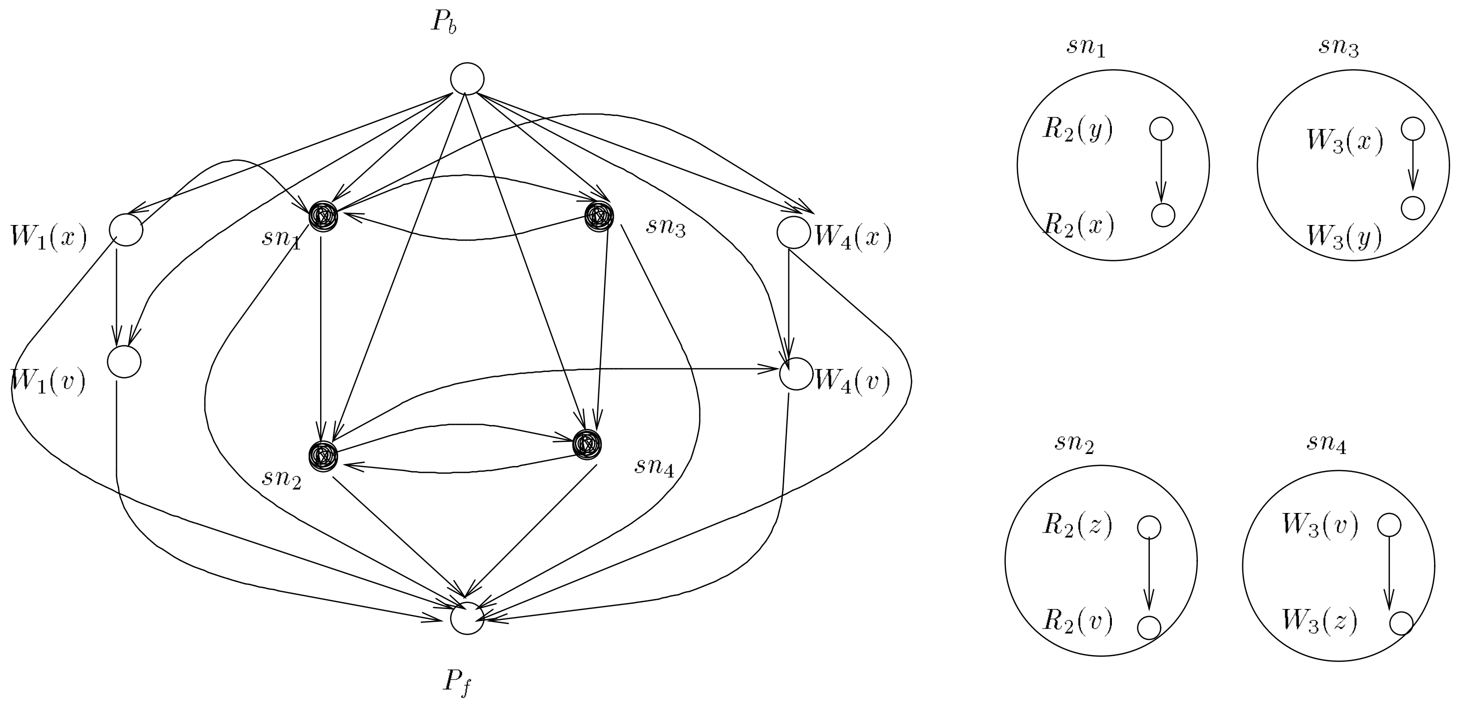

Figure 30: $\mathcal{H}(s_{13})$

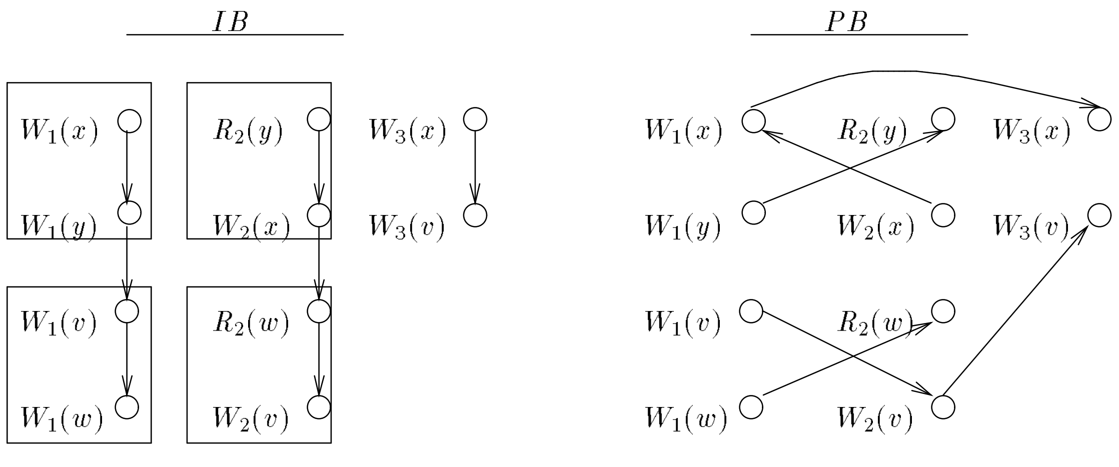

Figure 31: Execution s_{14}.

2. For each entity x and pair of operations $R_i^j(x)$ and $W_i^l(x)$ in atomic action j of process P_i, if $R_i^j(x)$ returns the value written by $W_i^l(x)$, then edge $(W_i^l(x), R_i^j(x))$ is added to E_i^j.

3. For each entity x and pair of operations $R_i^j(x)$ and $W_i^l(x)$ in atomic action j of process P_i, if $W_i^l(x)$ overwrites the value read by $R_i^j(x)$, then edge $(R_i^j(x), W_k^l(x))$ is added to E_i^j.

4. For each entity x and pair of operations $W_i^j(x)$ and $W_i^l(x)$, in atomic action j of process P_i, if $W_i^l(x)$ overwrites the value written by $W_i^j(x)$, then edge $(W_i^j(x), W_k^l(x))$ is added to E_i^j.

There are four types of arcs in E:

1. For each pair of operations a and b, if $parent(a) \neq parent(b)$, and a is before b in program order IB, and there is no other operation c such that $parent(a) \neq parent(c) \neq parent(b)$, a is before c and c is before b in IB, then the arc $(parent(a), parent(b))$ is added to E.

2. For each entity x and pair of operations $R_i^j(x)$ and $W_k^l(x)$, if $R_i^j(x)$ returns the value written by $W_k^l(x)$, and $parent(R_i^j(x)) \neq parent(W_k^l(x))$, then edge $(parent(W_k^l(x)), parent(R_i^j(x)))$ is added to E.

3. For each entity x and pair of operations $R_i^j(x)$ and $W_k^l(x)$, if $W_k^l(x)$ overwrites the value read by $R_i^j(x)$, and $parent(R_i^j(x)) \neq parent(W_k^l(x))$, then edge $(parent(R_i^j(x)), parent(W_k^l(x)))$ is added to E.

4. For each entity x and pair of operations $W_i^j(x)$ and $W_k^l(x)$, if $W_k^l(x)$ overwrites the value written by $W_i^j(x)$, and $parent(W_i^j(x)) \neq parent(W_k^l(x))$, then edge $(parent(W_i^j(x)), parent(W_k^l(x)))$ is added to E.

Theorem 7.5 *An execution s is conflict correct if and only if $\mathcal{G}(s)$ is acyclic.* □

In Figure 32, we show the higraph $\mathcal{G}(s_{14})$ corresponding to execution s_{14} in Figure 31. The higraph is cyclic, and therefore, the execution is not conflict correct. The problem of deciding whether an execution is conflict correct can be solved in polynomial time. Note that if an execution $s =< C, PB >$ does not contain any atomic actions, testing higraph $\mathcal{G}(s) = (V_1, V_0, E)$ becomes a regular graph $G(s) = (V_0, E)$. In this case, execution s is both conflict correct and conflict consistent.

8 Conclusions

An ideal system is one that performs program operations in the order specified by the program and executes atomic program segments exclusively. Although this system model simplifies the task of reasoning about both sequential and concurrent programs, its straightforward implementation yields poor performance. To enhance performance, concurrency and pipelining techniques can be used, which may result in data accesses that are performed in an order which is different from the order specified by the program, which may result in incorrect executions. An execution is correct if its result is equivalent to the result that could have been obtained had the execution taken place on the ideal system. In this paper, we have developed a unified general theory of

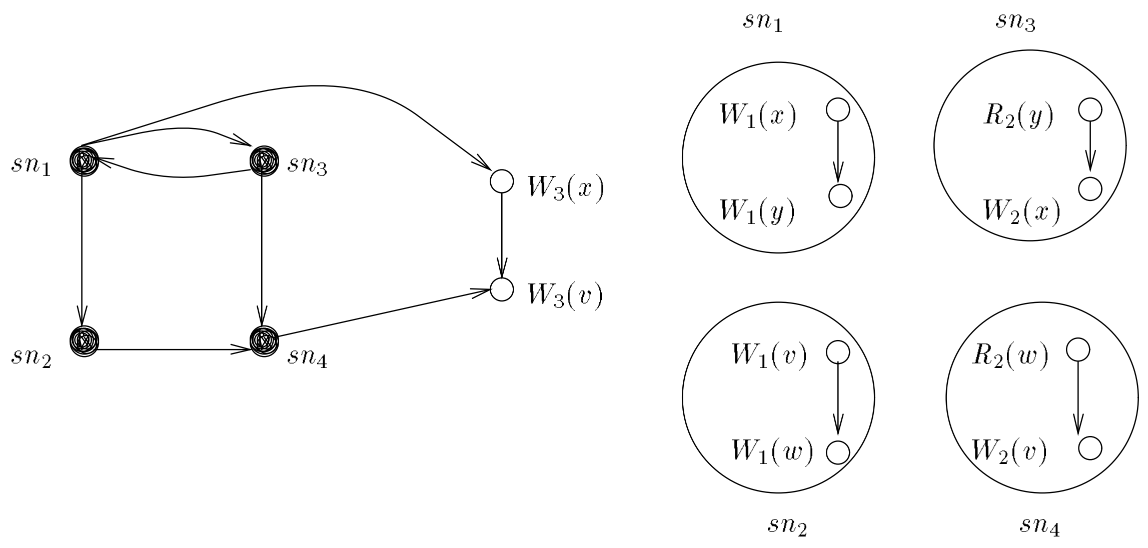

Figure 32: $\mathcal{G}(s_{14})$

correct executions where the access orders differ from the access order on the ideal system. Our unifying theory is applicable to a variety of programming paradigms, application domains, and architectures. It provides a verification tool to test the correctness of an execution, and allows us to devise more efficient protocols for various systems.

References

[1] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers,* September 1979, pp. 690-691.

[2] H. F. Korth, and A. Silberschatz, *Database System Concepts,* McGraw-Hill, 1991.

[3] A. Silberschatz, J. Peterson, and P. Galvin, *Operating System Concepts,* Addison-Wesley Publishing Company, Inc., 1991.

[4] G. R. Andrews, and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys,* June 1983, pp. 3-69.

[5] G. S. Almasi, and A. Gottlieb, *Highly Parallel Computing,* The Benjamin/Cummings Publishing Company, 1989.

[6] D. Shasha, and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Transactions on Programming Languages and Systems,* April 1988, pp. 282-312.

[7] C. Papadimitriou, *The Theory of Database Concurrency Control,* Computer Science Press, 1986.

[8] M. Dubois and C. Scheurich, "Memory access dependencies in shared-memory multiprocessors," *IEEE Transactions on the Software Engineering,* June 1990, pp. 660-673.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Computer Architecture News,* June 1990, pp. 15-26.

[10] S. V. Adve, and M. D. Hill, "Weak Ordering- A New Definition," *Proceedings of the 17th Annual International Symposium on Computer Architecture,* May 1990, pp. 2-14.

[11] P. Bitar, "The weakest memory-access order," *Journal of Parallel and Distributed Computing,* 1992, pp. 305-331.

[12] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, 1990.

[13] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 processor-memory element," *Proceedings of the International Conference on Parallel Processing,* 1985, pp. 782-789.

A Appendix

A.1 Proofs of Theorems in Section 5

Proof of Theorem 5.1:

s is B correct.

$\equiv$ {Definition 5.5 }

s is B equivalent to an execution s_x in X.

$\Rightarrow$ {Definition 5.4 }

s is view equivalent to s_x.

$\Rightarrow$ {Definition 5.2 }

s is view correct. □

Proof of Theorem 5.2:

s is conflict correct.

$\equiv$ {Definition 5.9 }

s is conflict equivalent to an execution s_x in X.

$\Rightarrow$ {Definition 5.8 }

s is B equivalent to s_x.

$\Rightarrow$ {Definition 5.5 }

s is B correct. □

A.2 Proofs of Theorems in Section 6

The following axioms directly follow from the definitions of higraphs and their underlying graphs Let $G = (V, E')$ be the underlying graph of higraph $\mathcal{G} = (V_1, V_0, E)$, and n_1, n_2 and n_3 three nodes in G, such that $parent(n_1) \neq parent(n_2) \neq parent(n_3) \neq parent(n_1)$ in $\mathcal{G}$.

Axiom A.1 *There is an edge (n_1, n_2) in G, if and only if there is an edge $(parent(n_1), parent(n_2))$ in $\mathcal{G}$.*

Axiom A.2 *There is a path between n_1 and n_2 in G, if and only if there is a path between $parent(n_1)$ and $parent(n_2)$ in $\mathcal{G}$.*

Axiom A.3 *There is a cycle through n_1 and n_2 in G, if and only if there is a cycle through $parent(n_1)$ and $parent(n_2)$ in $\mathcal{G}$.*

Axiom A.4 *If there is a cyclic supernode in $\mathcal{G}$, then G is cyclic.*

Axiom A.5 *If there is a supernode in $\mathcal{G}$ which is not totally ordered, then G is not totally ordered.*

Proof of Theorem 6.1:

For the only if direction:
$\mathcal{G}$ is acyclic.
$\equiv$ {Definition 6.5 }
G is acyclic.
$\Rightarrow$ {Axioms A.3 and A.4 }
G' is acyclic, and all supernodes are acyclic.

For the other direction:
Each supernode is acyclic, and G' is acyclic.
$\Rightarrow$ { Axiom A.3 }
If G has a cycle, it must be only through the nodes of which parent is the same in $\mathcal{G}$.
$\Rightarrow$ { Premise }
G is acyclic.
$\equiv$ {Definition 6.5 }
$\mathcal{G}$ is acyclic. □

A.3 Proofs of Theorems in Section 7

We introduce the following lemmas to simplify the proofs of theorems.

Lemma A.1 *If an execution s is in X, then $\mathcal{P}(s)$ is acyclic.*

Proof: Suppose s is in X. For s, we know that $PB \supseteq IB$ and PB is compatible with IB, and therefore the graph representation of PB is acyclic. We can build an acylic graph G as follows. The nodes of G are the nodes in O and two nodes P_b and P_f. G contains the graph representation of PB and has two additional directed edges (P_b, a) and (a, P_f) for each node a in O. Furthermore, if sn_i and sn_j are elements of O/A, such that $sn_i \; PB/A \; sn_j$, then for all nodes a and b such that $a \in sn_i$ and $b \in sn_j$, edge (a, b) is added to G. These edges do not generate cycles in G, since $PB/A \supseteq IB/A$ and PB/A is compatible with IB/A, and therefore the graph representation of PB/A is acyclic for s. G is the underlying graph of a higraph $\mathcal{G} = (V_1, V_0, E)$, such that $V_1 = AA$ and $V_0 = \{P_b, P_f\} \cup O - \bigcup_{S \in AA} S$. Then, E is the set G/A. $\mathcal{G}$ is acyclic. We claim now that $\mathcal{G}$ is compatible with hipolygraph $\mathcal{P}(s)$: Any arc in $\mathcal{P}(s)$ is certainly an arc in $\mathcal{G}$, and any arc in a supernode in $\mathcal{P}(s)$ is an arc in the same supernode in in $\mathcal{G}$, and for any choice (a, b, c) in $\mathcal{P}(s)$, either arc (a, b) or (b, c) is in $\mathcal{G}$, and for any choice (a, b, c) in a supernode in $\mathcal{P}(s)$, either arc (a, b) or (b, c) is in the same supernode in $\mathcal{G}$. $\mathcal{P}(s)$ is acyclic, since there is a compatible higraph $\mathcal{G}$ that is acyclic. □

Lemma A.2 *If s is view correct, then for any execution s_x in X, $\mathcal{P}(s) = \mathcal{P}(s_x)$.*

Proof: Suppose that $s =< C, PB >$ is view correct. This means that s is view equivalent to an execution $s_x =< C, PB_c >$ in X. Due to the definition of X (Definition 3.1), s_x has the same C as s. Hence, both executions s and s_x correspond to the same operations, program order, and atomic actions. Since s and s_x are view equivalent, in both executions, the read operations return the value written by the same write operations, initial values are returned by the same read operations, and final values are written by the same write operations (Definition 5.1). Thus, if all the steps for the construction of polygraphs $\mathcal{P}(s)$ and $\mathcal{P}(s_x)$ are followed, the same supernodes, nodes,edges and choices will be generated for both hipolygraphs. $\square$

Lemma A.3 *A higraph $\mathcal{G} = (V_1, V_0, E)$ is a total order, if and only if each supernode in V_1 is a total order, and graph $G' = (V_1 \cup V_0, E)$ is a total order.*

Proof:
For the only if direction:
$\mathcal{G}$ is a total order.
$\equiv$ {Definition 6.5 }
G is a total order.
$\Rightarrow$ {Axioms A.2 and A.5 }
G' is a total order, and all supernodes are a total order.

For the other direction:
Each supernode is a total order, and G' is a total order.
$\Rightarrow$ { Axiom A.2 }
If G is not totally ordered a cycle, it must be only because there is no edge among some pairs of nodes of which parent is the same in $\mathcal{G}$.
$\Rightarrow$ { Premise }
G is a total order.
$\equiv$ {Definition 6.5 }
$\mathcal{G}$ is a total order. $\square$

Proof of Theorem 7.1:

For the only if direction:
$s =< C, PB >$ is view correct.
$\Rightarrow$ { Lemma A.2 }
$\mathcal{P}(s) = \mathcal{P}(s_x)$.
$\Rightarrow$ { Lemma A.1 }
$\mathcal{P}(s)$ is acyclic.

For the other direction:
Suppose that $\mathcal{P}(s)$ is acyclic. Then, there is an acyclic directed higraph $\mathcal{G}$ which is compatible with $\mathcal{P}(s)$ (Definition 6.5). Higraph $\mathcal{G}$ can be completed to a total order $\mathcal{G}'$ in which P_b precedes all other operations and atomic actions in s, and P_f follows all other operations and atomic actions in s. Let G' be the underlying graph of $\mathcal{G}'$, and G'' be the subgraph of G' which excludes the nodes corresponding to P_b and P_f. Obviously, execution $s_x =< C, G'' >$ is in X. Now we will proof that s_x is view equivalent to s.

1. Suppose that $R_a^i(x)$ reads the initial value of entity x in s. Due to definition of $\mathcal{P}(s)$, for any write operation $W_c^k(x)$, there will be an edge $e = (R_a^i(x), W_c^k(x))$ in G''. Since G'' is acyclic and is the performing order of s_x, $R_a^i(x)$ reads the initial value of x in s, if and only if $R_a^i(x)$ reads the initial value of x in s_x.

2. Suppose that $W_a^i(x)$ writes the final value of entity x in s. Due to definition of $\mathcal{P}(s)$, for any other write operation $W_c^k(x)$, there will be an edge $e = (W_c^k(x), W_a^i(x))$ in G''. Since G'' is acyclic and the performing order of s_x, $W_a^i(x)$ writes the final value of x in s, if and only if $W_a^i(x)$ writes the final value of x in s_x.

3. Suppose that $R_d^j(x)$ returns the value written by $W_a^i(x)$ in s. Due to the definition of $\mathcal{P}(s)$, edge $e = (W_c^k(x), R_d^j(x))$ in G'. Furthermore, we claim that if G' is acyclic, there is no node $W_c^k(x)$, such that there is an edge $e_1 = (W_a^i(x), W_c^k(x))$ and an edge $e_2 = (W_c^k(x), R_d^j(x))$ in G'. Suppose G' is acyclic and has the edges e_1 and e_2. Since G' is the underlying graph of higraph $\mathcal{G}'$ that is compatible with $\mathcal{P}(s)$, G' must contain either the edge $(W_c^k(x), W_a^i(x))$ or the edge $(R_d^j(x), W_c^k(x))$. Then, there is a cycle in G'. Since G' is acyclic, G' cannot contain e_1 and e_2. Since G' is the performing order of s_x, $R_d^j(x)$ returns the value written by $W_a^i(x)$ in s, if and only if $R_d^j(x)$ returns the value written by $W_a^i(x)$ in s_x.

4. Since s and s_x consist of the same processes, write operations are dependent on the same entities. We assumed an architecture where a write operation is performed after the read operations on which the write operation is dependent. Hence, for each entity x and $W_i^j(x)$ that is dependent on entity y, $R_i^k(y)$ is the first read performed on y before $W_i^j(x)$ in execution s, if and only if $R_i^k(y)$ is the first read performed on y before $W_i^j(x)$ in execution s_x.

Hence, s is view equivalent to s_x, and therefore view correct. □

Lemma A.4 *A hipolygraph $\mathcal{P} = (V_1, V_0, E, C)$ is acyclic, if and only if each supernode $snp_i = (V_{1i}, E_{1i}, C_{1i})$ in V_1 is acyclic, and polygraph $P' = (V_1 \cup V_0, E, C)$ is acyclic.*

Proof:
Polygraph $P' = (V_1 \cup V_0, E, C)$ is acyclic, and each supernode in V_1 is acyclic.

$\equiv$
There is an acylic graph $G' = (V_1 \cup V_0, E')$ that is compatible with polygraph P', and for each supernode in V_1, there is an acylic graph $sn_i = (V_{1i}, E'_{1i})$ that is compatible with polygraph snp_i.
$\equiv$ { Theorem 6.1 }
Higraph $\mathcal{G} = (V_1, V_0, E')$ is acyclic.
$\equiv$
Hipolygraph $\mathcal{P} = (V_1, V_0, E, C)$ is acyclic. □

Theorem A.1 *The problem of deciding whether an execution is view consistent is NP-complete.* □

The proof of this theorem relies on the fact that determining whether a polygraph is acyclic, which is an NP-complete problem [7].

Proof of Theorem 7.2:

It follows from Theorem A.1 and Lemma A.4 that deciding whether a hipolygraph is acyclic is an NP-complete problem. Hence, the problem of deciding whether an execution is view correct is NP-complete. □

Lemma A.5 *If an execution s is in X, then $\mathcal{H}(s)$ is acyclic.*

Proof: Higraph $\mathcal{G}$ which is constructed in the proof of Lemma A.1 contains $\mathcal{H}(s)$. Thus, $\mathcal{H}(s)$ is acyclic. □

Lemma A.6 *If s is B correct, then for any execution s_x in X, $\mathcal{H}(s) = \mathcal{H}(s_x)$.*

Proof: Suppose that $s =< C, PB >$ is B correct. This means that s is B equivalent to an execution $s_x =< C, PB_c >$ in X. Due to the definition of X (Definition 3.1), s_x has the same C as s. Hence, both executions s and s_x correspond to the same operations, program order, and atomic actions. Since s and s_x are B equivalent, in both executions, the read operations return the value written by the same write operations, initial values are returned by the same read operations, final values are written by the same write operations, a write operation is performed before another write operation, if the value written by the second write operation is returned by a read operation, and if a write operation is performed after a read operation in s, this write operation is performed after the read operation in s_x (Definition 5.4). Thus, if all the steps for the construction of higraphs $\mathcal{H}(s)$ and $\mathcal{H}(s_x)$ are followed, the same supernodes, nodes and edges will be generated for both higraphs. □

Proof of Theorem 7.3:

For the only if direction:
$s =< C, PB >$ is B correct.

$\Rightarrow$ { Lemma A.6 }
$\mathcal{H}(s) = \mathcal{H}(s_x)$.
$\Rightarrow$ { Lemma A.5 }
$\mathcal{H}(s)$ is acyclic.

For the other direction:
Suppose that $\mathcal{H}(s)$ is acyclic. Higraph $\mathcal{H}(s)$ can be completed to a total order $\mathcal{G}$ in which P_b precedes all other operations and atomic actions in s, and P_f follows all other operations and atomic actions in s. Let G be the underlying graph of $\mathcal{G}$, and G' be the subgraph of G which excludes the nodes corresponding to P_b and P_f. Obviously, execution $s_x =< C, G' >$ is in X. Similar to proof of Theorem 7.1, one can go through each item in the definition of B equivalence (Definition 5.4), and prove that s_x is B equivalent to s, hence s is B correct. $\square$

Proof of Theorem 7.4:

The problem of deciding whether a graph is acyclic can be performed in $O(k^2)$ time, where k is the number of nodes in the graph. Thus, it follows from the Theorem 6.1 that we can test whether an execution is B correct in $O(n^2)$ time, where n is the total number operations in all processes. $\square$

Lemma A.7 *If an execution s is in X, then $\mathcal{G}(s)$ is acyclic.*

Proof: Suppose s is in X. For s, we know that $PB \supseteq IB$ and PB is compatible with IB, and therefore the graph representation of PB is acyclic. We can build an acylic graph G as follows. The nodes of G are the nodes in O. G contains the graph representation of PB and furthermore, if sn_i and sn_j are elements of O/A, such that $sn_i \; PB/A \; sn_j$, then for all nodes a and b such that $a \in sn_i$ and $b \in sn_j$, edge (a, b) is added to G. These edges do not generate cycles in G, since $PB/A \supseteq IB/A$ and PB/A is compatible with IB/A, and therefore the graph representation of PB/A is acyclic for s. G is the underlying graph of a higraph $\mathcal{G} = (V_1, V_0, E)$, such that $V_1 = AA$ and $V_0 = O - \bigcup_{S \in AA} S$. Then, E is the set G/A. $\mathcal{G}$ is acyclic. Obviously, $\mathcal{G}(s) = \mathcal{G}$. $\square$

Lemma A.8 *If s is conflict correct, then for any execution s_x in X, $\mathcal{G}(s) = \mathcal{G}(s_x)$.*

Proof: Suppose that $s =< C, PB >$ is conflict correct. This means that s is conflict equivalent to an execution $s_x =< C, PB_c >$ in X. Due to the definition of X (Definition 3.1), s_x has the same C as s. Hence, both executions s and s_x correspond to the same operations, program order, and atomic actions. Since s and s_x are conflict equivalent, in both executions, the conflicting executions are performed in the same order (Definition 5.8). Thus, higraphs $\mathcal{G}(s)$ and $\mathcal{G}(s_x)$ are the same. $\square$

Proof of Theorem 7.5:

For the only if direction:

$s =< C, PB >$ is conflict correct.

$\Rightarrow$ { Lemma A.8 }

$\mathcal{G}(s) = \mathcal{G}(s_x)$.

$\Rightarrow$ { Lemma A.7 }

$\mathcal{G}(s)$ is acyclic.

For the other direction:

Suppose that $\mathcal{G}(s)$ is acyclic. Higraph $\mathcal{G}(s)$ can be completed to a total order $\mathcal{G}'$. Let G' be the underlying graph of $\mathcal{G}'$, Obviously, execution $s_x =< C, G' >$ is in X. Similar to proof of Theorem 7.1, one can go through each item in the definition of conflict equivalence (Definition 5.8), and prove that s_x is conflict equivalent to s, hence s is conflict correct. $\square$